Full citation: Meldrum, S., Licorish, S. A., Owen, C. A. and Savarimuthu, B. T. 2020. Understanding stack overflow code quality: A recommendation of caution. Science of Computer Programming, 199. doi: 102516. 10.1016/j.scico.2020.102516

Understanding Stack Overflow Code Quality: A Recommendation of Caution

Sarah Meldrum, Sherlock A. Licorish*, Caitlin A. Owen and Bastin Tony Roy Savarimuthu
Department of Information Science
University of Otago
Dunedin, New Zealand
sarah-meldrum@outlook.com, sherlock.licorish@otago.ac.nz, oweca636@student.otago.ac.nz, tony.savarimuthu@otago.ac.nz

ABSTRACT

Community Question and Answer (CQA) platforms use the power of online groups to solve problems, or gain information. While these websites host useful information, it is critical that the details provided on these platforms are of high quality, and that users can trust the information. This is particularly necessary for software development, given the ubiquitous use of software across all sections of contemporary society. Stack Overflow is the leading CQA platform for programmers, with a community comprising over 10 million contributors. While research confirms the popularity of Stack Overflow, concerns have been raised about the quality of answers that are provided to questions on Stack Overflow. Code snippets often contained in these answers have been investigated; however, the quality of these artefacts remains unclear. This could be problematic for the software engineering community, as evidence has shown that Stack Overflow snippets are frequently used in both open source and commercial software. This research fills this gap by evaluating the quality of code snippets on Stack Overflow. We explored various aspects of code snippet quality, including reliability and conformance to programming rules, readability, performance and security. Outcomes show variation in the quality of Stack Overflow code snippets for the different dimensions; however, overall, quality issues in Stack Overflow snippets were not always severe. Vigilance is encouraged for those reusing Stack Overflow code snippets.

Keywords: Stack Overflow; Code Quality; Code Reliability and Conformance to Programming Rules; Code Readability; Code Performance; Code Security

*Corresponding author

1. INTRODUCTION

Community Question and Answer (CQA) platforms facilitate the use of the power of the crowd (i.e., online communities) to solve problems [32]. Platforms such as Stack Overflow and Yahoo!Answers[1] provide a service that benefits those who look to the internet to answer questions or find particular information [56]. In particular, such platforms benefit software practitioners seeking information, as it is likely that many other people have faced a similar problem, and so, a relevant question may have already been asked that has invoked a suitable answer. On the other hand, these platforms also allow new questions to be created, and experts can lend their specific experience, which allows them to solve a problem and gain the respect of their peers in the community.

While these websites host useful information, it is critical that the information provided on these platforms are of good quality, and that users can trust the information. This justifies a research agenda to afford this form of quality assurance. However, as this research area is still developing, even the terminologies used to identify such platforms are inconsistent. Srba and Bielikova [61] found that multiple terms are used for referring to CQA platforms. For example, while this paper refers to the types of community question and answer platforms as CQA, it is common for such platforms to be labelled Q&A, social Q&A and community forums [55, 57]. Beyond the terms used to identify CQA portals, Krüger, et al. [41] conducted a secondary study of CQA papers and suggest that the quality of questions and answers is a central challenge of CQA systems. In addition, Srba and Bielikova [61] completed a CQA survey and noted *preservation of long-term sustainability* as a key area for future research of these platforms. For CQA platforms to have long-term sustainability, however, establishing the quality of these sites is key to ensuring that users trust the platform and feel that they can continue to participate. Other approaches for encouraging sustainability of CQA platforms in inspiring user participation are to provide tools and employ gamification techniques [33].

Stack Overflow has been noted as a successful platform due to its user participation, suggesting that its employment of gamification techniques indeed contribute to its prominence [16]. However, there are questions surrounding the quality of answers on Stack Overflow [25, 64]. In addition, given that developers use many of the code snippets in posts on Stack Overflow during development [62], it is important to evaluate the quality of these artefacts. In particular, while the Stack Overflow community's collective surveillance may help to identify and improve errors in code snippets on this platform and users may appropriately use code snippets by adapting them to their specific problem/task, this is not always the case. Wu, et al. [68] examined how Stack Overflow code snippets were used in open-source projects and found that only 44% of the files containing Stack Overflow snippets showed that these snippets were modified prior to reuse by software developers. In fact, Bi [11] has shown that even the throwing/catching of generic exceptions is missed by software developers reusing Stack Overflow code snippets. We thus set out **to understand the quality of code snippets that are often provided in Stack Overflow posts**. The findings of this research could be helpful for directing future research on Stack Overflow, by facilitating investigations aimed at providing mechanisms to further scrutinise the aspects of quality that Stack Overflow code snippets do not meet. Additionally, developers selecting code snippets from Stack Overflow during software development will have a better understanding of the potential limitations of the code they use.

However, addressing the objective of this research project requires code snippet quality to be defined. This is done by assessing research and considering code snippet quality in relation to well established and understood software quality measurements [35]. This has led to our consideration of code reliability and conformance to programming rules, readability, performance and security (refer to Section 2.1 for discussion on this issue). Given our definition of quality, consisting of multiple dimensions, code snippets are extracted from Stack Overflow and analysed against these criteria. We then provide results at multiple levels of granularity, covering all violations (or errors), snippet specific violations and qualitative analysis assessing the implications for the presence of violations, and Stack Overflow community's efforts towards addressing these. The outcomes provided in this work are a survey of Stack Overflow code snippets' violations against these quality dimensions, as well as the types of violation that are evident in code provided by contributors. We also outline the basis for how the software development community may craft an agenda towards maintaining software quality, notwithstanding the utility that Stack Overflow provides.

[1] https://answers.yahoo.com

The remaining sections of this study are structured as follows. Section 2 considers the literature relating to Stack Overflow, and outlines subsequent research questions. In Section 3, the proposed methodology of this research is introduced and discussed. This leads to Section 4, which presents the results of the study in relation to the associated research questions. Section 5 provides a discussion of the results along with the implications. The threats of the research are next discussed in Section 6. Finally, Section 7 concludes the work and outlines potential areas for future research.

2. BACKGROUND AND RESEARCH QUESTIONS

In order to achieve the objective of this research, literature related to code quality on Stack Overflow is reviewed in Section 2.1, with particular emphasis on understanding works that have evaluated code. Thereafter, we synthesise the related literature to identify gaps and outline our research questions in Section 2.2.

2.1 Code Quality on Stack Overflow

Stack Overflow is the leading CQA platform for programmers, with more than 10 million users, contributing some 16 million questions as of 2019[2]. While few would doubt the utility of Stack Overflow and similar platforms to software development practitioners, there have been questions regarding the quality of the responses generated to practitioners' questions. For instance, in the quest to understand the quality of content on Stack Overflow, Ginsca and Popescu [25] investigated the relationship between Stack Overflow user profiles and answer quality, finding correlations between a more complete user profile and answer quality. Jin, et al. [33] studied how the need to 'win' reputation rewards can influence answer quality, as users may at times be driven to provide answers without considering quality implications. Anand and Ravichandran [5] identified a need for separating answer quality from members' popularity, as the reward system provided by Stack Overflow sometimes conflicts with quality answers. For example, users at times may try to answer questions relating to popular or easy topics, which in turn leads to an increase in their reputation. Additionally, studies such as Treude, et al. [64] and Asaduzzaman, et al. [6] have investigated how the quality of the question itself can affect the quality of the answer that is provided. These works all point to the need to be vigilant when approaching Q&A websites such as Stack Overflow for recommended solutions.

Given the preceding concerns, it is important for research work to help the software development community with quality validations of Stack Overflow to fill this gap. In addition, given that developers use many of the code snippets in posts on Stack Overflow during development [62], it is important to evaluate the quality of these artefacts. Some investigations have been conducted looking at specific aspects of code snippets, such as the work of Squire and Funkhouser [60], who recommended a 1:3 ratio of code to text in answers on Stack Overflow. The findings of these authors suggest that there is anticipation of a significant number of code snippets in answers provided on this portal. Acar, et al. [2] found in an experiment comparing various coding resources that the code sourced from Stack Overflow was not as secure as official documentation or books, indicating the need for improving code security. Yang, et al. [69] investigated the usability rates of code from Stack Overflow, finding only one percent (1%) of Java code extracted could be successfully compiled. These authors found an additional two percent (totalling 3%) of code could compile when adding class structures and semicolons to code in snippets that were missing these. While these papers investigate certain attributes of Stack Overflow code (i.e., security or usability), there is need for a comprehensive evaluation of code quality in Stack Overflow posts.

In fact, more recently, a number of studies have identified the use of Stack Overflow code snippets in open-source projects and analysed their effect on code quality. Ahmad and Ó Cinnéide [3] assessed GitHub projects, which have reused Java code snippets, in terms of code cohesion over time. They found that 42% of the project classes exhibited reduced cohesion, most of which did not regain the cohesion exhibited prior to adding the Stack Overflow code snippet. Abdalkareem, et al. [1] measured the quality of projects by classifying code commits as bug fixing or non-bug fixing. They found that the percentage of bug fixing commits in each project file was larger after adding the Stack Overflow code snippets. Ragkhitwetsagul, et al. [53] conducted a survey of Stack Overflow users, which found that common issues associated with Stack Overflow answers include outdated solutions and buggy code. This was confirmed when identifying Java snippets included in open-source projects, 66% of which were considered to be

---

[2] https://en.wikipedia.org/wiki/Stack_Overflow

outdated solutions and over 5% of which were considered to be buggy. Campos, et al. [14] extracted JavaScript snippets from Stack Overflow and analysed them in terms of code rule violations (using ESLinter),with the most common violation types relating to style issues (82.9% of the violations). A small number of the code snippets associated with violations were found to be used in GitHub projects. Nikolaidis, et al. [49] measured quality in terms of technical debt (the effort required to fix code inefficiencies), determining that the Java code snippets were actually associated with an overall lower technical debt density than the project code. However, there were a number of cases when the code snippets were associated with a much larger technical debt density than the project code.

Other studies have examined Stack Overflow code snippets for characteristics indirectly related to code quality. Treude and Robillard [65] assessed whether Java code snippets on Stack Overflow are self-explanatory. The answer text and code comments were removed from a sample of code snippets. They were presented to GitHub users, who determined that less than half of the code snippets were considered to be self-explanatory. The inappropriate use of a Stack Overflow snippet in a software development project, due to the snippet not being self-explanatory, is likely to reduce the quality of the project code. Therefore, examining the question and answers (including the user comments) associated with the code snippet may be needed in order for it to be self-explanatory. Wu, et al. [68] examined how Stack Overflow code snippets were used in open-source projects, finding that 44% of the files containing Stack Overflow snippets were modified prior to use in the file. This may be an indication of a lack of code quality associated with the original snippet. A survey of developers (who use Stack Overflow) found that 32% of respondents re-implemented code snippets rather than reuse them in their original state because of the perception that such code snippets may be of poor quality. Yang, et al. [70] also found that the exact duplication of code snippets was rare in the Python code projects examined.

When considering potential definitions of quality in relation to code (or software) this can be difficult to define due to different perceptions and expectations [35]. While studies mentioned above relate to code more generally, there are also studies which consider code quality explicitly. Jones and Bonsignour [35] look at multiple aspects of software quality such as technical quality, process quality and usage quality, totalling 121 quality attributes. While the study looks at software quality instead of code quality, these authors outline relevant aspects related to technical quality that could be of utility to this work. The ISO25010[3] standard defines software product quality as containing functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability and portability, with a number of sub-characteristics for each. Lu, et al. [44] refer to code quality as readability and maintainability, using tools to analyse aspects of quality such as potential bugs and comments in code. These authors also investigate quality in relation to casual and experienced users. Another study investigated code readability as a component of code quality, correlating these aspects with overall software quality [12].

In terms of Stack Overflow, Rahman, et al. [54] suggest answer comments can be used as a guide to evaluate the quality of code snippets. They infer that lower quality code snippets would result in more comments. In fact, debate around the quality of Stack Overflow questions and answers has echoed disagreements. For example, quality of questions was defined by a score of zero or more and the number of edits in one study [21], while answer quality was defined as those with scores of four or higher (13% of answers) by another study [48]. Considering these studies, it is clear that there are various components of code quality that are accepted by the software development community. How we respond to these metrics in providing a holistic and reliable view of Stack Overflow snippet quality is central to this study, an issue we consider in Section 3, having outlined our research questions in the following subsection.

2.2 Research Questions

To achieve the objective of this study, **to understand the quality of code snippets that are often provided in Stack Overflow posts,** it is important to consider the questions and concerns related to the quality of answers on Stack Overflow [25, 64], and the significance of the code snippets in these answers. In addition, it is important to establish what is meant by code snippet quality, given the variation in understandings and interpretations of quality [35]. That said, previous works have attempted to consider the measures that may influence or reduce Stack Overflow answer quality [25, 64]. Measures considered include: the degree to which Stack Overflow code compiles [69], code cohesion over time [3], the currency of snippet solutions [53], the effort required to fix code inefficiencies

[3] http://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[49], if code snippets are self-explanatory [65], if code snippets were reused directly [68], and the proportion of text and code in answers [60]. There has been less interest in investigating Stack Overflow snippet quality holistically. Given the relative lack of understanding of the quality of code snippets on Stack Overflow, we aim to provide a multi-dimensional view of code quality on this portal by answering the overarching question *what is the quality of code snippets provided in answers on Stack Overflow?*

Code snippet quality refers to the quality of the short snippets of code often contained in answers. This is in contrast to software quality, which assesses the quality of an overall software project codebase and related artefacts (including user interfaces, non-functional requirements, and so on) [35]. This extended view of code quality would not be reliably expected for code snippets provided in Stack Overflow posts, as these are often aimed at answering very specific coding concerns [26]. For example, in using User Datagram Protocol (UDP) as against Transmission Control Protocol (TCP), a user may be interested in understanding if there is a server socket for datagrams in Java, as is the case for the normal server socket (i.e., is it possible to use the "DatagramServerSocket" constructor having imported the "java.net" package). Recommendations provided in response for such a query may thus be brief, with little conformance to coding conventions used when writing large volumes of code. Accordingly, in this study, code snippet quality is defined as having the dimensions of reliability and conformance to programming rules, readability, performance and security, which are further discussed in Section 3.1.

Given this definition of Stack Overflow code snippet quality, we break down our overarching question into the following four research questions:

RQ1. What is the <u>reliability</u> and <u>conformance to programming rules</u> of code snippets provided in answers on Stack Overflow?

RQ2. What is the <u>readability</u> of code snippets provided in answers on Stack Overflow?

RQ3. What is the <u>performance</u> of code snippets provided in answers on Stack Overflow?

RQ4. What is the <u>security</u> of code snippets provided in answers on Stack Overflow?

3. METHODOLOGY

In line with the open nature of our overarching question, w*hat is the quality of code snippets provided in answers on Stack Overflow?,* we employ an exploratory tone to our analyses, starting with an exhaustive body of quantitative analysis before performing deeper qualitative analysis. This later analysis considers implications for quality violations, and Stack Overflow community's effort towards ensuring contributors' awareness of potential code quality shortcomings. We provide further details on the two forms of analyses (quantitative and qualitative) below. First, we further contextualise code snippet quality (in Section 3.1), before then considering code quality on Stack Overflow (in Section 3.2). The latter subsection outlines the techniques that are adapted to answer RQ1–RQ4, while the former subsection focusses on further defining code quality.

3.1 Code Snippet Quality Criteria

Code quality is difficult to define [35], and as such it is important to define what is meant by code snippet quality in this study. In reviewing the literature, it is clear that code quality consists of many dimensions [21, 35]. To establish a set of code quality attributes in assessing software quality, code quality and code snippet quality are considered both individually and in relation to each other. When trying to define the quality of code snippets, considering software quality at a higher-level can be important as this topic is more widely defined and understood. In addition, in considering code snippet quality, it is assumed that these snippets will be copied and pasted into a larger project (i.e., are a subset of code). Therefore, an analysis of the literature was performed to understand the various dimensions of code snippet quality. Note also that it is assumed that Stack Overflow code snippets are intended to be correct (i.e., not purposefully wrong).

First, considering software quality, Jones and Bonsignour [35] look at software quality from an economical perspective, going on to define seven categories of software quality: technical/structural quality, process quality, usage quality, service quality, aesthetic quality, standards quality and legal quality. A more specific and acceptable

view of software quality is ISO25010[4], which defines software product quality in terms of eight characteristics – functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability and portability. Similarly, Spinellis [59] and Zou, et al. [71] recommend the six software quality attributes as defined by the earlier ISO/IEC9126 standard: functionality, reliability, usability, efficiency, maintainability, and portability; with security included under functionality, and performance efficiency included under efficiency. The future cost of coding shortcuts and defects released in code (technical debt) is also now considered integral to code quality [19]. All dimensions of code quality are linked to technical debt, including defects that are related to functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability and portability [23]. These defects may lead to costs associated with loss of opportunity, liability and revision effort [31]. In addition, Kottom [39] defines five metrics that are important for measuring code quality, particularly pertaining to software testing. These are: *code coverage* – how much of the code is tested; *security* – ensuring code mitigates potential vulnerabilities; *performance and efficiency* – understanding how long a program takes to perform particular tasks; *style and complexity* – having a unified style and commenting to ensure code can be read by other developers; and finally, *Lines of Code (LOC)* – in particular, lines of code per 'something' to understand the complexity of the application (e.g., lines of code per bug).

On the basis that many of the quality guidelines do not provide common and direct metrics for measuring code quality, efforts have also been dedicated to providing practical quality models. For instance, the maintainability index (MI) was proposed for providing a measure of source code maintainability [50]. The SIG maintainability model is said to address the shortcomings of the MI by moving beyond the provision of a measure to provide cues around specific characteristics of maintainability that are deficient in software [28]. The delta maintainability model (DMM) extends the SIG maintainability model providing maintainability measurements at the commit level [20]. Quamoco is also said to close the quality characteristics and measurement gap [66]. Other attempts at the provision of concrete solutions include: Squale [47], CAST which attempts to infer technical debt [15], SQALE [42], and ColumbusQM [8].

With Stack Overflow providing a source of coding help for less knowledgeable developers, it is plausible that defects in code snippets may propagate when these are reused unknowingly. More knowledgeable developers may also make trade-off decisions between long-term code quality and short-term gains in delivering rapid releases (e.g., due to market pressure or the need to increase productivity), which may lead to the accumulation of technical debt [23, 30]. That said, while functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability and portability factors may be useful for measuring code snippet quality, these factors are aimed at assessing an entire software project or packaged software, which means they are not entirely relevant to the small snippets of code on Stack Overflow. In fact, attempting to use this exhaustive list of measures would also not be realistic (e.g., it would be unfair to measure the *maintainability* of Stack Overflow code snippets given that code on this portal often targets a specific user's question or concern – see example provided in Section 2.2). Given this reality, it is important to understand Stack Overflow code snippets at a lower level.

Given the evidence in the general works on software quality above, six potential code snippet quality dimensions were initially considered in this work. These were: *reliability*, *security*, *maintainability/readability*, *usability*, *compatibility* and *performance*. Readability was chosen over maintainability because maintainability has been defined by ISO25010 as consisting of modularity, reusability, analysability, modifiability and testability. While these components are appropriate for measuring software quality more generally, they would be inappropriate for assessing code snippets. This is because code snippets are assumed to be a subset of code only, and therefore assessing the maintainability of the snippet code from a perspective of readability is more appropriate. That said, such artefacts would lend themselves to checks for conformance to programming best practices (or programming rules). Programming rules are related to reliability, given the potential for all software faults to influence software failures at a later stage (i.e., if/when programming constructs are confusing and error prone) [58, pp. 298-299]. Summerville notes that reliability deals with system faults or errors, where failure may range in severity, from crashes to minor errors [58, pp. 296-297]. It was also noted that reliable systems should conform to specification. If the intent of Stack Overflow contributors writing code snippet is to provide correct solutions, such solutions should be free of errors and conform to good programming conventions to be considered reliable. Usability was excluded

[4] http://iso25000.com/index.php/en/iso-25000-standards/iso-25010

because there was overlap with the reliability dimension, while compatibility was removed because it is difficult to define whether code snippets are compatible with the question asked without performing complex text mining analysis. Investigations of such scale are outside the scope of this paper. Furthermore, it is not feasible to establish the specific code standards that were the target or intention of contributors of code snippets, making it unfair to evaluate compatibility in relation to software versions. This evaluation left the remaining four code quality attributes (**reliability and conformance to programming rules**, **readability**, **performance** and **security**) as the dimensions that are suitable for measuring code snippet quality as described in Table 1. Detailed assessments of these dimensions are provided in Section 3.2.4.

Table 1. Code snippet quality dimensions

Quality Dimension	Description
Reliability and Conformance to Programming Rules (RQ1)	Code snippets should not be broken, confusing or prone to runtime errors; meaning that they should be able to compile (given appropriate syntactic adjustments) and contain no bugs or errors. Code snippets should also conform to generally accepted programming rules. This can be measured by attempting to compile and run the code snippets, capturing the specific failure or violation (refer to Section 3.2.4).
Readability (RQ2)	Code snippets should follow standard Java readability conventions to ensure code can be easily understood and maintained in the future. This can be measured by checking if code snippets meet such conventions and whether they are explained (refer to Section 3.2.4).
Performance (RQ3)	Code snippets should consider the performance or efficiency of proposed solutions. For example, has the code snippet provided an answer to the question in a way that saves processing or reduces the number of processing steps (refer to Section 3.2.4)?
Security (RQ4)	Code snippets should consider security constraints of proposed solutions, so as not to compromise security. For example, using interpolated strings instead of prepared statements for database input would be considered a security error (refer to Section 3.2.4).

3.2 Code Quality on Stack Overflow

Not much is known about the quality of code on Stack Overflow, and so, insights into this phenomenon will help to inform the software development community's actions around the use of this platform, and also generate hypotheses for subsequent testing. To provide these contributions we must evaluate code snippets extracted from Stack Overflow against the definition of code snippet quality, as described in the previous section. Thus, in this subsection we provide details around how data was extracted from Stack Overflow in Section 3.2.1. In Section 3.2.2 we discuss the selection of tools to assess the given quality criteria, and how these tools are tested in a pilot study in Section 3.2.3. We then define the final tool analysis process for this study in Section 3.2.4, and explain our qualitative analysis process in Section 3.2.5.

3.2.1 Data Extraction and Analysis

Data was extracted from Stack Overflow using Stack Overflow's data explorer[5] which allows data to be queried using standard SQL Statements. Answer posts which contained at least one "<code>" tag and were from a question tagged with Java were then sampled. This sample has been chosen due to Java's popularity[6]. In addition, we know that Stack Overflow data are generalizable across languages and time [43, 70] and, therefore, we anticipated that our sample would provide transferable insights relating to Java code on Stack Overflow for the software development community.

We next needed to appropriately identify relevant code snippets. The "<code>" tags indicate the presence of a code snippet, as Stack Overflow's answers are structured in HTML style, with code snippets included between code tags (i.e., <code> </code>); see Figure 1.b for example. Finally, answers for 2014, 2015 and 2016 were chosen as these years had the highest number of questions and answers on Stack Overflow, and we have studied and confirmed a good level of code reuse that is evident by the software engineering community for these snippets [43]. Thus, it was fitting to now understand how wholesome these snippets are given their frequent reuse. This resulted in our dataset comprising 117,526 answers (46,103 accepted answers and 71,423 unaccepted answers). We also extracted the other attributes pertinent to the questions that generated these answers, and our dataset was imported into a Microsoft SQL Server database.

[5] https://data.stackexchange.com/stackoverflow/query/new

[6] Based on RedMonk programming language popularity: http://redmonk.com/sogrady/2016/02/19/language-rankings-1-16/

The Java code was then extracted from the answerBody of the answer post using a small Java program, with each code snippet between the tags "<code> … </code>" being saved separately. During our exploration of Stack Overflow code, we observed that there were in-line code snippets (surrounded by <code> tags) and code blocks (surrounded by <pre><code> tags). Closer checks of these aspects revealed that in-line code snippets were typically added as part of text answers when contributors needed to provide explanation (refer to "&&" highlighted in Figure 1), while code blocks provided more substantial implementation of specific solutions (refer to Figure 1.b). Accordingly, we felt that it would be unrealistic and unfair to analyse the quality of in-line code snippets against our quality dimensions, and thus excluded these from our sample. This resulted in 404,779 code snippets, from the 117,526 answer posts (191,556 snippets from accepted answers and 213,223 snippets from unaccepted answers). A brief explanation and summary of this data is shown in Appendix A.

Considering this research project's proposed analysis and the works of Duijn, et al. [21] and Yang, et al. [69], it was decided that only code snippets with one or more lines of code (i.e., contained a new line character) would be analysed for quality. In particular, Yang, et al. [69] found that by excluding single word Java code snippets 6.2% of code snippets were parsable (up from 3.9%). Therefore, it was important to exclude at a minimum, code snippets that did not contain a new line character such as single word Java code snippets. The snippets were extracted from the database using a Java application which checked if the code snippets contained 'import', 'package' or 'class', and then saved the file as is. If these words were not present these files were encased in a public class structure, in line with Yang, et al. [69], as shown in Figure 2 (i.e., the first code snippet featured in Figure 1). Each of these files were saved as a .java file with a unique identifier name (e.g., C1234.java represented code snippet number 1234) so that results could be traced back to each code snippet.

We were cautious not to manipulate the code snippets extensively to risk confounding out results and analysis, thus, we did not attempt to conduct extensive repairs on code snippets (e.g., importing packages, deleting spaces, adding ";", and so on). In fact, our *minimal* encasing of code snippets resulted in errors, which we have accounted for in our analysis below. Exploring these outcomes, we anticipated that additional manipulation of code snippets in order to increase our pool of Stack Overflow code snippets for analyses would have adversely affected the reliability of our outcomes. However, we plan to perform follow up snippet repair studies. Our processing allowed 151,954 of the code snippets from 94,279 answers to be used for our analyses (i.e., 37.5% of the code snippets). This sample comprises 66,389 snippets from accepted answers and 85,565 snippets from unaccepted answers, allowing us to compare the quality of snippets across these two groups. Attributes included in our dataset are answer identification number (*answerId*), question identification number (*questionId*), answer score (*answerScore*), answer creation date (*answerCreationDate*), answer body (*answerBody*), question date (*questionDate*), question score (*questionScore*), view count (*ViewCount*), answer count (*AnswerCount*), and comment count (*CommentCount*). We also computed *LOC*[7], *Code Length*[8], *Code Spaces*[9] and *SPA*[10] to aid our analyses (refer to Appendix A for descriptions of all our data attributes and summary statistics for our dataset).

We did not include the history of post edits in our quantitative analysis as the Stack Overflow history logs largely include details unrelated to code (e.g., title history, tags history, owner history, and so on), and these fields were 'null' for most posts. That said, our deeper qualitative analysis considers all historical data related to the posts. The properties of the 151,954 code snippets that were processed for analysis are visualised in Figure 3 (Figure 3.b shown using a log scale for the x and y axes). Figure 3.a shows that answers in our sample frequently had one or two snippets. It is also observed in Figure 3.b that the *LOC* for the parsed and unparsed code snippets exhibit similar distributions, with the majority of code snippets in each group including between 4 and 60 *LOC*. This similarity in distribution observed here (in Figure 3.b) is promising in terms of the extrapolation of our findings to those code snippets that did not parse due to errors (further details are provided below).

[7] Refers to the number of lines of code.
[8] Refers to the length of the code snippet in terms of number of characters.
[9] Refers to the number of space characters (i.e., ' ') in the code snippet provided as a part of answers.
[10] Refers to the number of code snippets per answer associated with each code snippet.

3.2.2 Tool Selection

Considering related research, three possible tools were found for analysis of the code snippets in helping to understand the four attributes of code snippet quality above. These tools are PMD[11], Checkstyle[12] and FindBugs[13], as they are able to evaluate Java code and have been used to good effect in this regard in previous research [7, 17, 21]. These tools have also been accepted by the software engineering community, and particularly given the growing momentum of 'open-source' static analysis tools due to their accessibility when compared to 'closed-source' alternatives [34]. These tools were initially assessed for appropriateness for this study by considering the checks they perform in relation to the code snippet quality attributes, and understanding how they perform these checks. Reliability and conformance to programming rules are assessed by both PMD and FindBugs, and in fact both of these tools are able to detect common programming flaws and bugs. However, PMD requires code snippets to be in .java files, while FindBugs requires code to be compiled (.class files). Readability is assessed by Checkstyle as it is used to check that code meets coding standards, such as the Google and Sun[14] standards, including aspects such as the structure and comments of code. There are two main limitations evident in prior work which quantifies readability by computing a readability score for code (ranging from 0 to 1) [13]. First, the work does not explain which readability features are missing in the code. Second, only a limited feature set (containing 19 features) is considered to compute the readability score. In contrast, Checkstyle used in our work considers three times as many readability features (i.e., 56 checks; refer to Section 3.2.4), and also identifies which of these features are violated by the code, which addresses the purpose of the work.

Performance is assessed by specific categories of checks in both FindBugs and PMD, which include 29 and 21 checks respectively. Security is assessed by two of FindBugs's categories of bug checks, which are *security* and *malicious code vulnerability*, including a total of 28 checks. Each tool has its own particular assumptions and limitations and these are discussed in Section 6.

[11] https://pmd.github.io

[12] http://checkstyle.sourceforge.net

[13] http://findbugs.sourceforge.net/index.html

[14] Note that while this is also known as the Oracle style, the standard was last updated by Sun Microsystems in 1999: http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html

a. Example Stack Overflow Answer

The right style is:

```
int random = (int)(Math.random() * 100);
```

Second:

your if structure is terrible:

for example:

```
if(guess >= 100 || guess <= 0){
```

1. you said guess is greater and equal 100
2. you said guess is less and equal 0
3. you used || operand instead of `&&` to check the number to be in range of 0 and 100

try this :

```
if (guess <= 100 && guess >= 0) {
        if (guess > random) {
            System.out.println("The guess is too high.");
        } else if (guess < random) {
            System.out.println("The guess is too low.");
        } else if (guess == random) {
            System.out.println("YOU WIN");
        }
    } else {
        System.out.println("Error, that is not a number between 0-100.");
    }
```

Explanation: if the gussing number is between 100 and 0, do the following. if it is not tell the user that he or she is out of the range.

b. Associated HTML Code Tags

```
<pre><code>int random = (int)(Math.random() * 100);
</code></pre>
```

```
<pre><code>if(guess >= 100 || guess <= 0){
</code></pre>
```

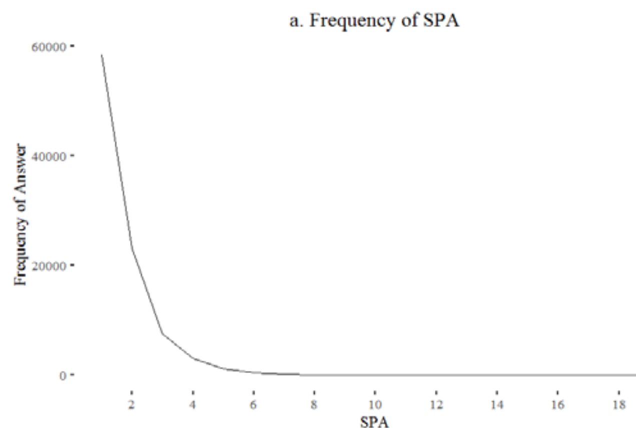
```
<li>you used || operand instead of <code>&&</code> to check the number to be in range of 0 and 100</li>
```

```
<pre><code>if (guess <= 100 && guess >= 0) {
    if (guess > random) {
      System.out.println("The guess is too high.");
    } else if (guess < random) {
      System.out.println("The guess is too low.");
    } else if (guess == random) {
      System.out.println("YOU WIN");
    }
  } else {
    System.out.println("Error, that is not a number between 0-100.");
  }
</code></pre>
```

Figure 1. Example Stack Overflow Answer (a) and Associated HTML Code Tags (b)

```
1 public class C281596{
2 int random = (int)(Math.random() * 100);
3
4 }
```

Figure 2. Example of code snippet being encased in a public class structure



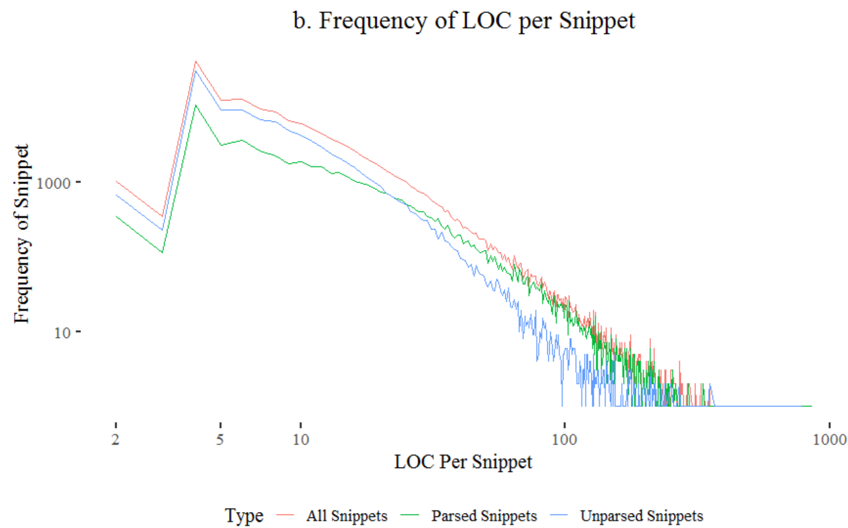


Figure 3. Frequency of SPA (a) and Frequency of LOC per Snippet (b)

3.2.3 Code Snippet Pilot

To check the appropriateness of PMD, Checkstyle and FindBugs, a pilot study was conducted on 100 code snippets. Initially, PMD and Checkstyle were used to analyse 100 code snippets as they both required .java files as inputs. Outcomes show that PMD was able to capture the unparsable files, and list them as errors in the output. These errors were then used to determine the code snippet files that should be removed for Checkstyle analysis, as Checkstyle would not execute with unparsable files. This meant that PMD had to be scheduled to execute first in order to find the unparsable files, so that these could be removed for Checkstyle. This also left a smaller subset of files to be compiled for FindBugs. We are cognisant that this reflects a limitation of the work, and this issue is considered at length in Section 6. Of the 100 code snippets that were analysed 35 could be analysed by PMD and Checkstyle, as they were able to be parsed and constructed into an Abstract Syntax Tree[15]. Even though these 35 code snippets were parsable (see Figure 4 for examples of snippets that caused errors), only nine could be compiled and therefore analysed by FindBugs. The result output of FindBugs includes summary information about the bugs found in individual files as well as the details of each individual bug found. From this process, we noticed that files which had not been edited (i.e., contained 'import', 'class' or 'package') could not be compiled due to the class name and file name (which was saved as a unique identifier) being different. Therefore, to overcome this issue in recovering as many files as possible for analysis "public class" was replaced with "class" in these files in an attempt to bypass this rule. Other reasons for parsing and compilation errors included missing brackets, quote marks, packages and import statements. There were also instances of unexpected return values (for methods with a void return type) and the absence of a main class. As noted above, we were very cautious not to manipulate the code snippets to risk confounding out results and analysis, and so we decided against trying to repair the snippets to increase our code pool. However, on examining some of the files that were not parsable, we observe that indeed the content therein was not useful programming code, instead at times some snippets included pseudo code or documentation as a way of explaining coding concepts (refer to Figure 4). Accordingly, employing heuristics to fix such code snippets would not be a trivial exercise.

In ensuring reliability of the outcomes, the first two authors informally checked the generated output for 100 random errors (50 for each author) that were returned by the PMD, Checkstyle and FindBugs tools against the actual code snippets from where the violations were derived (e.g., several 'Coupling' errors were returned by PMD). These checks showed that the 100 errors were all traceable in the actual code snippets. We were thus satisfied that the tools were suitable for our planned analyses. We performed the full quantitative analysis (via the PMD, Checkstyle and FindBugs tools), then the second and third authors performed two rounds of formal reliability checks involving

[15] https://www.techopedia.com/definition/22431/abstract-syntax-tree-ast

deeper qualitative analysis (refer to Section 3.2.5).

```
1 public class C100946{
2 hue
3 The parameters 'hue' and 'sat' are used to set the colour
4 The 'hue' parameter has the range 0-65535 so represents approximately
5 182*degrees (technically 182.04 but the difference is imperceptible)
6
7 }
```

```
1 public class C51056{
2 int data = 0;
3 ...
4 do{
5     ...
6 } while (data != 0);
7
8 }
```

Figure 4. Examples of snippets that caused parsing errors

3.2.4 Final Tool Analysis Process

Given the results of the 100 code snippet pilot study, a final processing pipeline was determined as shown in Figure 5. The 151,954 code snippets extracted from the database were first analysed by PMD to determine the number of parsable files, so these could be removed before Checkstyle could be executed. This subset of files were then compiled, for the FindBugs tool to analyse. For each of these tools, the frequency of broken checks or bugs found, in addition to the types of checks/bugs found, were analysed. Outputs of all tools are available in XML format; an example of each output is shown in Appendix C. A detailed explanation of how each quality dimension is assessed is provided below, while a summary of the tools used and the code quality dimensions assessed by these tools are shown in Table 2.

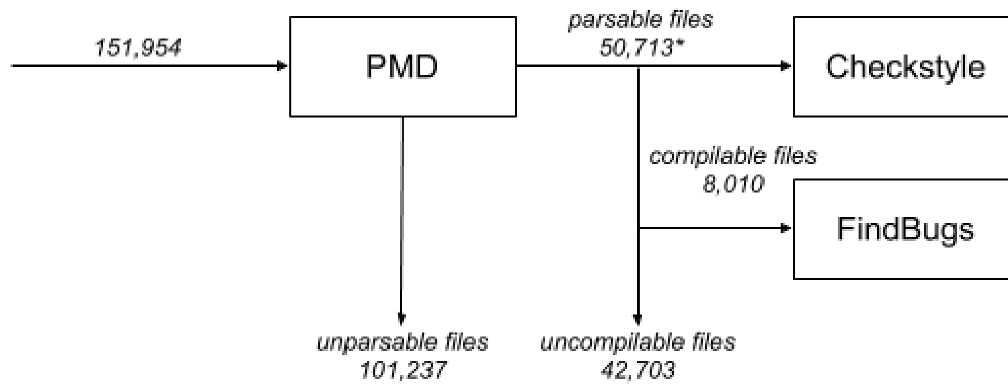


Figure 5. Process for code snippet analysis

Reliability and Conformance to Programming Rules (RQ1): Code snippets should not be broken, confusing or prone to runtime errors; meaning that they should be able to compile (given appropriate syntactic adjustments) and contain no bugs or errors. Code snippets should also conform to generally accepted programming rules. This was assessed by conducting checks using PMD, to determine the number of checks that have not been met (i.e., checks that are violated). The checks that were chosen to be assessed initially included all PMD categories of checks; however, summary and subjective checks were excluded as they do not assess reliability and conformance to programming rules (e.g., 'controversial' and 'code size' categories). These checks were clearly identified by the tool, where the first two authors assessed their relevance to code reliability and conformance to programming rules and agreed that they were not relevant (multiple rounds of formal reliability checks were subsequently conducted, refer to Section 3.2.5). This totals 10 context categories, with a total of 82 individual reliability checks and 138 individual conformance to programming rules checks, as listed in Appendix D.1. Each of the context categories was added to an XML file as required for input for the tool. In addition to understanding the number of checks violated per code snippet, and the categories of violations, the priorities of violations can be assessed from one (highest priority – 'change absolutely required') to four (lowest priority – 'change optional'). For transparency, rigour and completeness, we report on all violations that were detected, before closely examining those violations that are relevant to code snippets, and then providing further outcomes for deeper qualitative analysis. This pattern of reporting is provided for all four categories of violations (i.e., code reliability and conformance to programming rules, readability, performance and security).

Table 2. Tools used to assess code snippet quality attributes

Quality Attribute	How it is assessed (tool)	Number of checks
Reliability and Conformance to Programming Rules	PMD	220
Readability	Checkstyle – Google checks	56
Performance	FindBugs (*performance* category)	29
Security	FindBugs (*security* and *malicious code vulnerability* categories)	28

Readability (RQ2): Code snippets should follow standard Java readability conventions to ensure code can be easily understood and maintained in the future. This was assessed by using Checkstyle to check against Google's Java style conventions, to see how many of these conventions are not met by Stack Overflow code snippets. Google's Java style was chosen over the Sun's Java style due to its detailed guide[16], and Checkstyle's coverage of the standard[17] was adapted. It is also assumed that these styles largely duplicate each other, and thus, by conforming to Sun's Java style, contributors to Stack Overflow will also largely satisfy Google's Java style (and vice versa). In addition, Checkstyle supplies the relevant input file *google_checks.xml*[18], which assesses the structure of the files in line with the Google Java style. This file also details the 56 checks that are assessed (see D.2 in Appendix D).

Performance (RQ3): Code snippets should consider the performance or efficiency of proposed solutions. This was assessed by using the FindBugs tool, and assessing the *performance* category of bugs, to see how many have been found, totalling 29 bug checks. These are detailed in Appendix D (see D.3). By default, FindBugs searches for all bugs, so *performance* results are extracted from this larger dataset. Results are assessed in relation to the number of violations per code snippet and the types (categories) of bug violations found.

Security (RQ4): Code snippets should consider security constraints of proposed solutions, so as not to compromise security. This was assessed by using the FindBugs tool, and assessing the *security* and *malicious code vulnerability* categories of bugs, to see how many have been found, totalling 28 checks, as detailed in Appendix D (see D.4). As *performance* and *security* are assessed by the same tool, FindBugs, they are processed and analysed in the same way.

While wrapping the code snippet in a class declaration was necessary for analysing some of the code snippets, as noted above, this caused nine types of reliability and conformance to programming rules and readability errors identified by the tools described in Section 3.2.2 (see Table 3). In Table 3, the *IndentationCheck* errors occurred because the class wrapper was added without indenting the original code. The *WhitespaceAroundCheck* errors occurred because the class declaration wrapper did not include a space between the <codeIdentifier> and the open curly bracket (refer to Figure 2). The *CloneMethodReturnTypeMustMatchClassName* and *ProperLogger* errors occurred because the class name was the code snippet unique identifier rather than a name used within the snippet. The code snippets that already included class declarations were saved in Java files named after the code snippet unique identifier. This caused an *OuterTypeFilenameCheck* error as the file name did not match the class name. The rest of the errors were caused by the omission of a keyword in the class declaration. Table 3 shows that altogether, nine types of violations were caused by our data pre-processing, with these varying between 0.93% (*CommentsIndentationCheck*) and 100% (*OuterTypeFilenameCheck*) of the total number of errors of those types that were found in the Stack Overflow code snippets.

Note that not all incidences of the nine types of errors in Table 3 were caused by our pre-processing steps. This detail is provided in Table 3 (in the '% of Total Number of Violations' column). It is noted in Table 3 that only the 'OuterTypeFilenameCheck' errors were entirely caused by our pre-processing steps (see the '100%' entered in the '% of Total Number of Violations' column). Some of the code snippets that were not manipulated by us reported many incidences of the other eight types of errors when they were analysed. For instance, the 'ClassWithOnlyPrivateConstructorsShouldBeFinal' errors were largely evident in other Stack Overflow snippets that were not manipulated (i.e., only 16.33% of this type of error were caused by our pre-processing steps). Examples of the errors caused by the class declaration wrapper and Java file names are shown in Appendix B. The specific errors that were caused by our pre-processing steps were all removed prior to subsequent analysis and reporting of the outcomes, which are provided in Sections 4.1.1, 4.2.1, 4.3.1 and 4.4.1.

---

[16] https://google.github.io/styleguide/javaguide.html

[17] http://checkstyle.sourceforge.net/google_style.html

[18] https://github.com/checkstyle/checkstyle/blob/master/src/main/resources/google_checks.xml

Table 3. Errors caused by class declaration and Java file names (pre-processing steps)

Violation Name	Quality Dimension	Tool	Cause of Violation	Number of Violations	% of Total Number of Violations
ClassWithOnlyPrivateConstructorsShouldBeFinal	Reliability and Conformance to Programming Rules	PMD	A class with private constructors is not 'final'.	32	16.33
CloneMethodMustImplementCloneable	Reliability and Conformance to Programming Rules	PMD	The 'clone' method is implemented by a class that does not implement the 'Cloneable' interface.	7	36.84
CloneMethodReturnTypeMustMatchClassName	Reliability and Conformance to Programming Rules	PMD	The return type of the 'clone' method does not match the class name.	7	38.89
ProperLogger	Reliability and Conformance to Programming Rules	PMD	A logger is not associated with the correct class name.	2	25.00
UseUtilityClass	Reliability and Conformance to Programming Rules	PMD	A class only has static methods but is not a utility/abstract class.	5775	53.91
CommentsIndentationCheck	Readability	Checkstyle	A comment is incorrectly indented.	15	0.93
IndentationCheck	Readability	Checkstyle	A line of code is incorrectly indented.	63026	15.68
OuterTypeFilenameCheck	Readability	Checkstyle	The class name does not match the file name.	14396	100.00
WhitespaceAroundCheck	Readability	Checkstyle	The open curly bracket associated with the class declaration is not surrounded by whitespace.	34710	29.82

3.2.5 *Qualitative Analysis*

While the PMD, Checkstyle and FindBugs tools allowed us to conduct an exhaustive body of quantitative analysis, it was necessary to perform deeper qualitative analysis to triangulate these outcomes, thus making our insights more meaningful to the software engineering community. This later analysis was largely intended to reliably assess code snippets and consider implications for quality violations and Stack Overflow community's effort towards ensuring contributors' awareness of potential code quality shortcomings. We performed two phases of qualitative analyses, as detailed below.

Violations Analysis (phase 1): We checked all types of violations returned by the tools for their relevance to code snippets. Since code snippets on Stack Overflow are embedded in answers that provide context for their use, it was necessary to determine the *violations that would be fairly attributed to code snippets*. We explored all types of violations that were returned by the tools (reliability and conformance to programming rules=191, readability=50, performance=25 and security=14), identified irrelevant violations and removed them from our analysis. The following violation characteristics were considered as not being relevant to code snippets:

- Violations relating to the lack of particular types of content in the code snippet, which do not prevent a program from running (e.g. comments, Javadoc tags).
- Violations relating to Java file names (as a code snippet is not directly associated with a Java file).
- Violations relating to the inclusion of code that would not cause a runtime error if removed.

Table 4 provides a summary of the violations that were deemed irrelevant to code snippets. In arriving at the decision to tag the three violations characteristics above as irrelevant, each violation (of the total 280 violations) was selected by the second and third authors and discussed, with arguments put forward for 'inclusion/exclusion' and debated in a 'first round' of evaluation (using a bottom up process). If there was agreement between the authors the violation was flagged *true*, or *false* otherwise. Violations which attracted a false flag were discussed in a second round of evaluations, where final agreement was recorded (*true* or *false*). We used these outcomes to compute inter-rater agreement as measured using Holsti's coefficient of reliability measurement (C.R.) [29], which revealed a 96.7% agreement initially (first round) and then 100% subsequently (second round). This represents excellent agreement between evaluators, suggesting that a consistent and reliable approach was taken. The violations that were included as relevant to code snippets are reported in Sections 4.1.2, 4.2.2, 4.3.2 and 4.4.2.

Table 4. Violations (errors) that were deemed irrelevant to code snippets

Violation Name	Quality Dimension	Tool	Cause of Error	Number of Errors
UncommentedEmptyConstructor	Reliability and Conformance to Programming Rules	PMD	An empty constructor is not associated with a comment.	468
UncommentedEmptyMethodBody	Reliability and Conformance to Programming Rules	PMD	An empty method is not associated with a comment.	1061
UnusedPrivateField	Reliability and Conformance to Programming Rules	PMD	A private field is declared but not used.	3727
UnusedPrivateMethod	Reliability and Conformance to Programming Rules	PMD	A private method is declared but not used.	1676
UnusedImports	Reliability and Conformance to Programming Rules	PMD	An import statement is included in the code but not used.	1274
JavadocMethodCheck	Readability	Checkstyle	A method is missing a Javadoc comment.	19595
OneTopLevelClassCheck	Readability	Checkstyle	Each top-level 'class', 'interface' or 'enum' should be in its own source file.	1724
Upm_Uncalled_Private Method	Performance	FindBugs	A private method is declared but not used.	272
Urf_Unread_Field	Performance	FindBugs	A field is assigned a value but not read.	2508
Uuf_Unused_Field	Performance	FindBugs	A field is declared but not used/assigned a value.	607

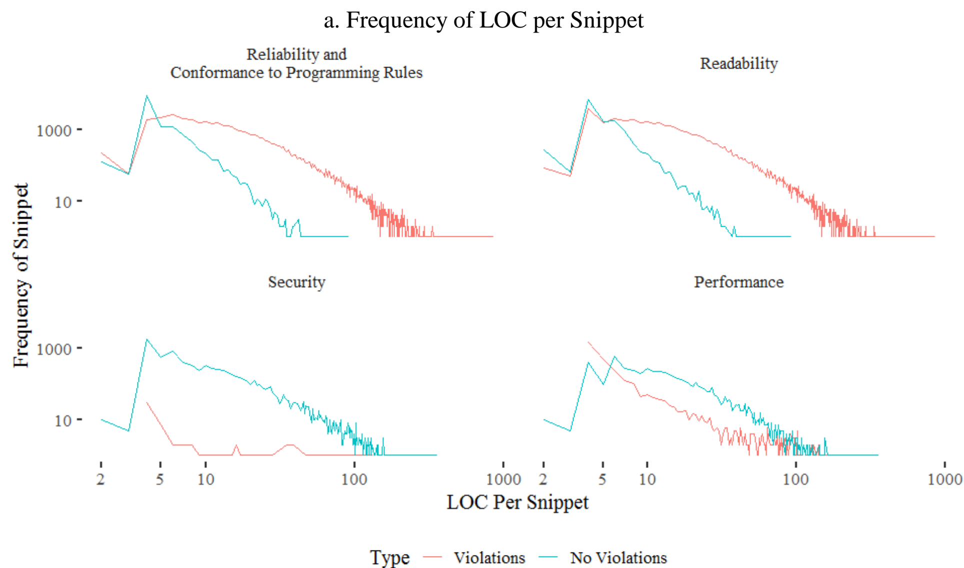
Snippet Analysis (phase 2): We conducted a second round of deeper qualitative analysis at the snippet level. We used *purposive sampling*, a common approach used in qualitative studies [45] to select a sample based on a specific purpose or strategy. In our work, the strategy was to select at least one code snippet for each specific type of violation within all the four broader categories of violations (see all the different types of violations in Appendix D). Purposive sampling thus enabled manual scrutiny and evaluation of *all* types of violations, in so doing, accommodating heterogeneous violation types (i.e., our work employs *heterogeneity sampling* a specific sub-type of purposive sampling [63]). Note that purposive sampling is the most common type of sampling technique used in the software engineering community (out of 10 sampling techniques), with 61% of the studies (145 out of 236) employing this technique [4]. Our analysis included 60 code snippets that returned violations for each of the four code quality categories (i.e., reliability and conformance to programming rules, readability, performance and security), thus a total of 240 snippets were considered. The number 60 is chosen for each category in our study because it accommodates for the largest number of specific types of violations in the four categories rounded to the nearest ten (i.e., the readability category has 56 different types of violations, and this number 56 is rounded to 60). Note that the sample size considered is not only representative of the different types of violations (as indicated above), but also compares with what is considered in the domain of code snippet analysis (e.g., 100 snippets were studied in Baltes, et al. [9], 136 in Campos, et al. [14], and 200 in Duijn, et al. [21]).

To conduct the analysis, the second and third authors first jointly determined the number of instances for each violation in order to sample at least one code snippet for each error and sample code snippets proportional to the number of instances of an error. Once samples were extracted (60 each), code snippets were studied. We determined the line numbers associated with the error as identified by the tool and then confirmed that the violation was not a false positive. We subsequently studied all of the Stack Overflow artefacts that were related to that code snippet (question, answers, comments, histories, and so on). Here, tracing from the question, we explored all the contextual information that was provided by contributors to determine whether there was evidence that the context of the code snippet would prevent the error from propagating to developers' code if the snippet was reused. For instance, we studied the 'text/answer' around the code snippet that described whether the code was taken from the question code that should be changed, whether the code should be added to the question code or whether the code is meant to explain a programming concept or method (i.e., not intended for a specific program). We also explored the potential that the answer may directly discuss the error featured in the sampled snippet and its implications. To support this evaluation, we studied the 'comments/responses' and 'histories' (including code edits) to ascertain the presence of a solution to the error or awareness that the error exists among the Stack Overflow community members. If the context around a code snippet makes it clear that the error is unlikely to propagate if the code was reused, we recorded this

information. We next studied the possible implications of the error not being fixed based on Java programming guidelines[19]. In ensuring a reliable inductive process, the second and third authors jointly developed the process and worked through one snippet for each category of violation (i.e., reliability and conformance to programming rules, readability, performance and security) in ensuring the process was rigorous. The third author then analysed 240 Stack Overflow code snippets independently (60 for each category of violation), before the second author performed a counter analysis of 100 of these snippets (25 for each category of violation). Differences in viewpoints were recorded by the second author, before formal discussions ensued for each difference noted. Overall, there were seven differences observed initially (93% agreement); however, these were all resolved on consensus (resulting in 100% agreement) [29]. Outcomes from these deeper analysis are reported in Sections 4.1.3, 4.2.3, 4.3.3 and 4.4.3.

4. RESULTS

In Figure 6, we visualise the general patterns of outcomes for violations that were found by the PMD, Checkstyle and FindBugs tools (excluding those violations that were introduced (during pre-processing) by the class declaration wrapper or code snippet unique identifier file name). As seen in Figure 3, the results in Figure 6 show that the code snippets that produce errors and the code snippets that do not produce errors have somewhat similar properties (refer to Figure 6.a). The majority of the code snippets that do not produce *reliability and conformance to programming rules* errors consist of between four and 10 LOC, and the majority of code snippets that do produce *reliability and conformance to programming rules* errors consist of greater than four LOC (refer to Figure 6.a). The pattern of outcomes is also convergent for *performance violations*, where the majority of the code snippets that do not produce *performance* errors and those that do produce *performance* errors consist of greater than four LOC. The majority of both the code snippets that do not produce errors and code snippets that produce errors consist of greater than four LOC for *readability* and *security* errors. Figure 6.b also shows that longer code snippets are more likely to produce errors and a larger number of errors, particularly for *reliability and conformance to programming rules* and *readability* errors.



[19] https://checkstyle.sourceforge.io/apidocs/, https://pmd.github.io/pmd-5.8.1/pmd-java/rules/index.html, http://findbugs.sourceforge.net/bugDescriptions.html

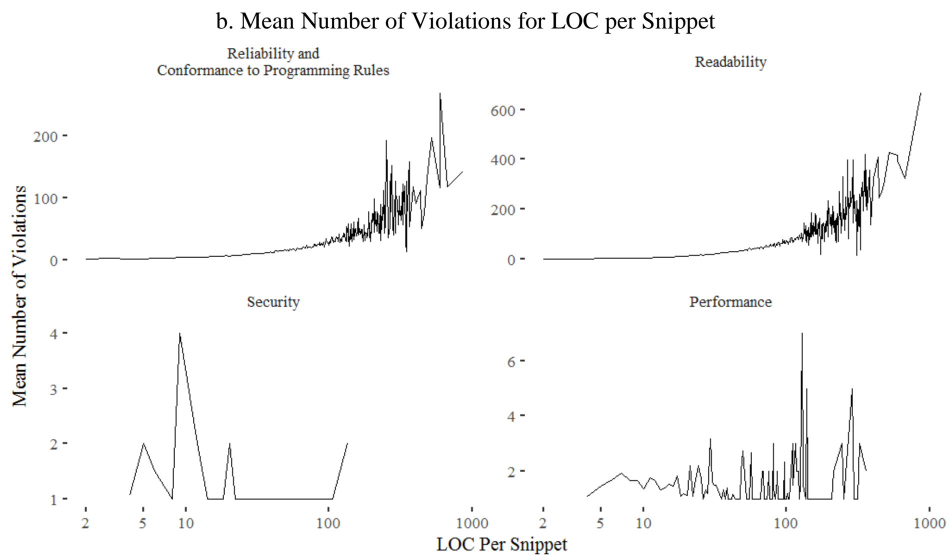


Figure 6. Frequency of LOC per snippet (a) and Mean number of violations for LOC per snippet (b)

In order to answer the research questions, RQ1–RQ4, the results are divided into four subsections, Sections 4.1 to 4.4. For these research questions and code snippet quality dimensions, both an understanding of how often the checks were broken (or number of violations found) and the types of checks/violations found are presented. We also report on outcomes for the snippet relevant violations and qualitative analysis. Section 4.5 then provides a summary of these results, including additional correlational evaluations between the quality dimensions and a range of code snippet variables, in line with our exploratory stance. We thereby look to evaluate other potential attributes that may impact the pattern of results observed in Sections 4.1 to 4.4, as another form of triangulation.

Analysing the initial 151,954 code snippets, 50,717 were analysed for reliability and conformance to programming rules using PMD, but only 50,472 were assessed for readability using Checkstyle due to errors (i.e., related to UTF-8 parsing issues). A further 8,010 code snippets were assessed for performance and security using FindBugs. This pipeline is summarised in Figure 5, which also depicts the number of unparsable and uncompilable files that were encountered. A summary of these results is shown in Table 5, considering the number of violations per code snippet. We also provide examples of code violations throughout the qualitative results in Sections 4.1.3, 4.2.3, 4.3.3 and 4.4.3. Finally, the correlation of variables was assessed for each quality dimension using Spearman's rank correlation (*rho*) given that the distributions violated the normality assumption. Our code snippet dataset is available for follow up analysis here: https://tinyurl.com/uv8jtfq. Please include the full citation to this study if you reuse this dataset.

4.1 Reliability and Conformance to Programming Rules (RQ1)

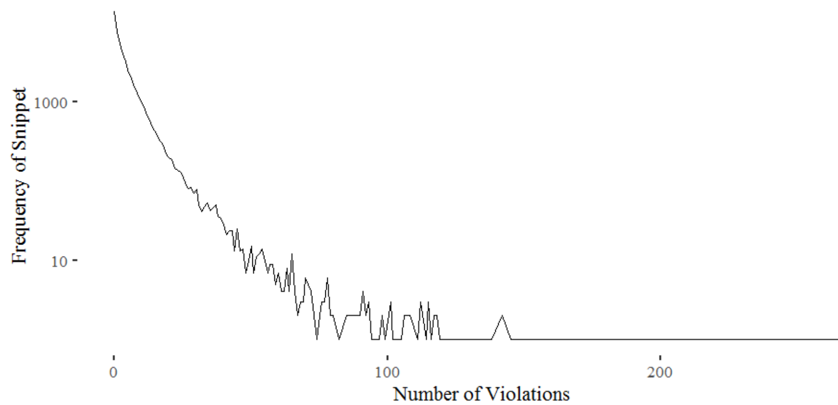
4.1.1 All Reliability and Conformance to Programming Rules Violations

As noted in Section 3, from the 151,954 included code snippets, 50,717 (33.38%) were parsable and could be analysed by the PMD tool. This left 101,237 code snippets which the tool was unable to parse due to an "error while parsing" or "error while processing". We are aware that this represents a limitation of the study, and have addressed this issue at length in Section 6. The parsable code snippets were analysed and resulted in 244,266 violations from 36,810 (72.58%) code snippets. On average, there were 4.82 violations per code snippet, with the minimum number of violations per code snippet being zero, and the maximum number being 269. The code snippet associated with 269 violations has *LOC*=597. The answer containing the snippet (for Question ID 26843289) is rated poorly by Stack Overflow users, having a score of only 0. Figure 7.a shows that the majority of the code snippets contain a small number of violations, while a small number of code snippets contain a large number of violations. Figure 7.b shows the same data grouped into bins of five to provide a more compressed visual representation of the data. This shows that 62.81% (23,121) of snippets with violations had between one and five violations, 19.99% (7359) had between six and 10 violations, 8.11% (2987) had between 11 and 15 violations, 3.72% (1370) had between 16 and 20 violations, and only 5.36% (1,973) had more than 20 violations. Categorised by priority in Figure 8, 1.22% (285/23,316) of reliability violations had the highest priority of violation, 0.60% (139) were priority two, the majority of violations (98.16%, or 22,886) were priority three and 0.03% (6) were priority four. For conformance to programming rules violations, 0.29% (638/220,950) had the highest priority of violation, 9.98% (22,058) were priority two, the majority of violations (87.29%, or 192,858) were priority three and 2.44% (5396) were priority four

(refer to Figure 8). Table 6 displays the number of violations as categorised by violation category from PMD. Here it is shown that the *code style* category represents the majority of violations with 109,553, followed by *design* with 57,083 violations, *best practices* with 42,717 violations and *reliability* with 23,316 violations. Table 6 reveals that the least prominent violations belonged to the *Jakarta Commons logging* category (with only 39 violations). These violations varied across priorities; however, on balance the most common categories are primarily associated with priority three violations. We next examine those violations with particular relevance to code snippets in detail.

Table 5. Summary of violations per code snippet for the four quality attributes

Number of Violations	Reliability and Conformance to Programming Rules	Readability	Performance	Security
Minimum	0	0	0	0
Lower Quartile	0	0	0	0
Median	2	3	0	0
Average	4.82	10.51	0.49	0.01
Upper Quartile	6	12	1	0
Maximum	269	667	18	4
No. of Code Snippets analysed	***50717***	***50472***	***8010***	***8010***



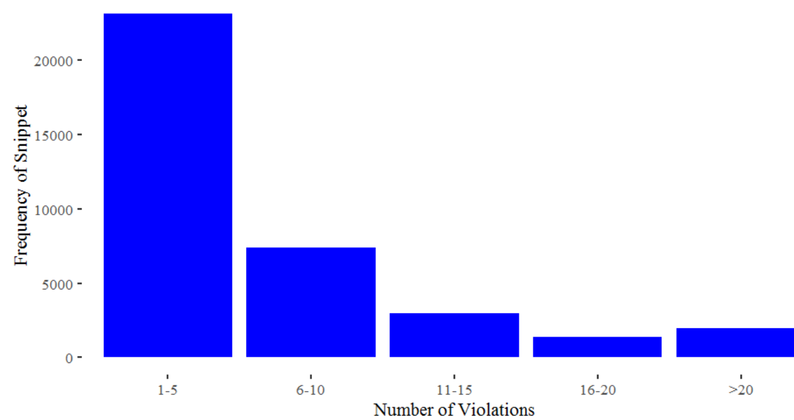


Figure 7. Number of reliability and conformance to programming rules violations per snippet (a) and Number of reliability and conformance to programming rules violations grouped (b)

4.1.2 Snippet Relevant Violations

Of the 186 reliability and conformance to programming rules violations considered to be relevant to code snippets, the 20% most relevant violations (38 violations) for code snippets were determined. As the PMD tool assigns a priority to each violation, the violation types were ranked in order of priority (starting with the highest priority violations) followed by number of violations (see Table 7). The most relevant violation was

AvoidThrowingRawExceptionTypes, which is associated with a high priority because using a raw exception type can make the cause of an error unclear and, therefore, lead to debugging difficulties. Also, the violation is equally relevant to both code snippets and Java files because throwing an exception is often required and managing errors is an important part of the functionality of a program. That said, this violation was not observed to be very frequent in Stack Overflow code snippets (261 times or 0.12%). Other relevant reliability and conformance to programming rules violations sorted by priority (criticality) are provided in Table 7, which shows various incidences, with some being particularly prominent (e.g., *MethodArgumentCouldBeFinal*, *LocalVariableCouldBeFinal*, *LawOfDemeter* and *SystemPrintln*). The least relevant violation included in the top 20% of violations was *AvoidInstantiatingObjectsInLoops* because although it is associated with a large number of violations (1,959), performing a computationally expensive operation is considered to be of lower priority.

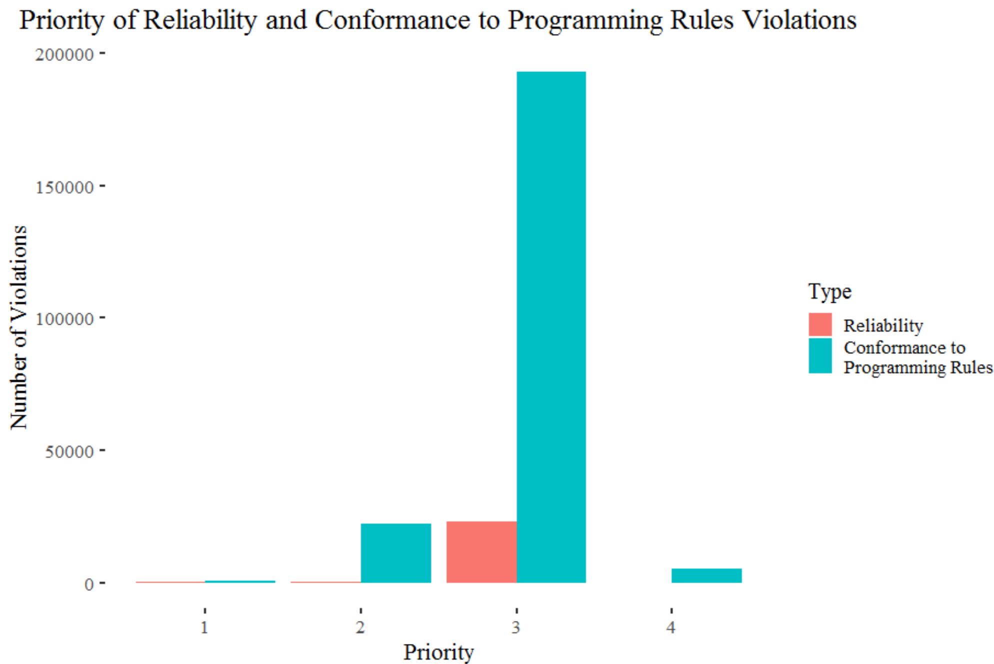


Figure 8. Priority of reliability and conformance to programming rules violations

Table 6. Number of reliability and conformance to programming rules violations per category

Rules violated by category	Priority	Number of Violations	Percentage of Violations
Code Style	1,3,4	109553	44.85%
Design	1,3	57083	23.37%
Best Practices	2,3,4	42717	17.49%
Reliability	1,2,3,4	23316	9.55%
Performance	2,3	6620	2.71%
Multithreading	1,3,4	3274	1.34%
Documentation	3	1529	0.63%
Additional rulesets	3	83	0.03%
Java Logging	2	52	0.02%
Jakarta Commons Logging	3	39	0.02%
Total		***244266***	

Table 7. Reliability and conformance to programming rules violations most relevant to code snippets

Rule Violated	Priority	Number of Violations	Percentage of Violations	Rule Violated	Priority	Number of Violations	Percentage of Violations
AvoidThrowingRaw ExceptionTypes	1	291	0.12	SingletonClass ReturningNew Instance*	2	12	< 0.01
ConstructorCalls OverridableMethod*	1	248	0.10	AvoidLosing Exception Information*	2	7	< 0.01
ClassWithOnlyPrivate ConstructorsShould	1	163	0.07	SingleMethod Singleton*	2	6	< 0.01

BeFinal							
EmptyMethodIn AbstractClassShould BeAbstract	1	84	0.03	MoreThanOne Logger*	2	4	< 0.01
AbstractClassWithout AnyMethod	1	52	0.02	Short Instantiation	2	4	< 0.01
AvoidThrowingNull PointerException	1	47	0.02	ProperClone Implementation*	2	4	< 0.01
ReturnEmptyArray RatherThanNull*	1	37	0.02	BrokenNull Check*	2	1	< 0.01
DoubleChecked Locking	1	1	< 0.01	Byte Instantiation	2	1	< 0.01
SystemPrintln	2	20068	8.22	MethodArgument CouldBeFinal	3	53930	22.08
AvoidReassigning Parameters	2	1301	0.53	LocalVariable CouldBeFinal	3	49194	20.14
IntegerInstantiation	2	271	0.11	LawOfDemeter	3	42505	17.40
StringInstantiation	2	193	0.08	BeanMembers ShouldSerialize*	3	17913	7.33
GuardLogStatement	2	127	0.05	UseUtilityClass	3	4938	2.02
GuardLogStatement JavaUtil	2	52	0.02	ImmutableField	3	3910	1.60
LoggerIsNotStatic Final*	2	47	0.02	AccessorMethod Generation	3	3026	1.24
AvoidBranchingState mentAsLastInLoop*	2	43	0.02	DoNotUse Threads	3	2785	1.14
BooleanInstantiation	2	24	0.01	UnusedLocal Variable	3	2315	0.95
LongInstantiation	2	17	0.01	AvoidPrint StackTrace	3	2102	0.86
AvoidMultipleUnary Operators*	2	15	0.01	Avoid Instantiating ObjectsInLoops	3	1959	0.80

***Reliability violations**

4.1.3 Qualitative Analysis

As noted in Section 3.2.5, we sampled 60 code snippets that returned reliability and conformance to programming rules violations for deeper qualitative analysis. With a goal to shed more light on the severity and seriousness (or lack thereof) of these violations, we sampled 30 snippets from the priority one category (highest priority – 'change absolutely required') and 30 from the priority four category (lowest priority – 'change optional') for analysis. We sampled all categories of violations for these two priorities, exploring the rule that was violated, the implication for breaking the rule (for code reuse) and context that is provided around the snippets (e.g., in question, answers, comments) which may lead to fixing the violation or an awareness that it exists.

Table 8 provides a summary of our outcomes for the 30 priority one violations. Here it is shown that eight different violations were evident in the code snippets, with some awareness among the community for three types of violations or 37.5% (*AbstractClassWithoutAnyMethod*, *DoubleCheckedLocking*, *EmptyMethodInAbstractClass-ShouldBeAbstract*). This evidence is promising as the violations point to the Stack Overflow community's awareness of coding standards related to proper inheritance behaviour of classes, the performance of objects and appropriate use of classes. For example, the 'singletonInstance' object should be 'volatile' or a nested static class could be used to implement lazy initialisation. While the code snippet in Figure 9.a ('SingletonClass' class, Question ID 23721115) and accompanying text do not make reference to 'double-checked locking', the issues associated with this snippet are made clear by the comments and rating (-1) that were generated by the answer. All of the comments state that the 'singletonInstance' *object needs to be volatile*. If the question author or a developer copied the code snippet for reuse, they are likely to edit the snippet based on the comments that were provided.

Less awareness was observed for five other types (62.5%) of violations in Table 8 (*AvoidThrowingNullPointerException*, *AvoidThrowingRawExceptionTypes*, *ClassWithOnlyPrivateConstructors-ShouldBeFinal*, *ConstructorCallsOverridableMethod*, *ReturnEmptyArrayRatherThanNull*). These code reliability and conformance to programming rules violations relate to proper error handling (and especially for helping with debugging), inconsistency in class design and usage and handling of nulls and potential exceptions. Throwing

'NullPointerExceptions' could be fixed by including an error message which is recognisable as a programmer-initiated exception, thus making the cause of the error clear. A class that is not 'final' (i.e., can be extended) and has only 'private constructors' (invocation can only occur inside the class) may lead to subclasses being created when it is assumed that this is not the case (by others), leading to potential security challenges. A constructor calling an 'overridable method' may cause it to be invoked on an object which has not been entirely constructed, leading to debugging difficulties. Returning a 'null reference' may cause an unexpected 'NullPointerException', and such references require null checks making code complicated and unreadable. An example of this lack of awareness (or oversight) by the Stack Overflow community is seen for Question ID 27386421 (refer to Figure 9.b). The author included '*//etc*' as a comment in the code block, which makes it clear that more conditional statements need to be used in order to return '*neighbours*'. However, if all of the required conditional statements are not used, the "*getNeighbor*" method may return a null reference. There is no evidence in any of the artefacts for this question (ID 27386421) to suggest that members of the Stack Overflow community observed this violation.

a. Answer Associated with Question ID 23721115

Following is a sample of singlton class,

-1

```
public class SingletonClass {
    private static SingletonClass singletonInstance = null;

    private SingletonClass() {
    }

    public static SingletonClass getSingletonInstance() {
        if(singletonInstance == null) {
            synchronized (SingletonClass.class) {
                if(singletonInstance == null) {
                  singletonInstance = new SingletonClass();
                }
            }
        }
        return singletonInstance;
    }
}
```

This is wrong. The way the code is, it only works in multithreaded SINGLE core CPU systems. In multicore system this method is faulty. There is no "happens-before" link between the writes and reads. – Oct 8 '16 at 17:31

Make singletonInstance volatile to guarantee happens-before – Nov 17 '16 at 6:42

Yeah...needed a object to be volatile, private volatile static SingletonClass singletonInstance = null; – Jan 13 '17 at 7:57

As discussed before, `volatile` is required. – Oct 12 '17 at 16:43

b. Code Snippet Associated with Question ID 27386421

```
28 public int[] getNeigbor(int side) {
29     if(side == 0) {
30         return neighbors[0];
31     }
32     //etc
33     return null;
34 }
```

Figure 9. Answer associated with Question ID 23721115 (a) and Code snippet associated with Question ID 27386421 (b)

We observed a similar mixed pattern of outcomes for the 30 code snippets with less serious priority one reliability and conformance to programming rules violations (lowest priority – 'change optional'). Of 10 different types of violations, evidence showed that there was context provided around the snippets (e.g., in question, answers, comments) which may lead to fixing four types of violations (40%) or an awareness that these exist (e.g., for *ExtendsObject* and *DontCallThreadRun*). On the other hand, there was no such evidence for six types (60%) of violations (e.g., *UselessParentheses* and *DuplicateImports*). While these violations may not be assessed seriously, they could make code expressions hard to read and increase the size of code and the chance of unwanted dependencies when a package is no longer being used. It would thus be prudent for the Stack Overflow community to guard against them.

Table 8. Qualitative analysis for priority one (highest priority) reliability and conformance to programming rules violations

Rule	Implications of Breaking Rule	Question IDs	Context Would Fix Error (%)
AbstractClassWithout AnyMethod	An abstract class designed to only store data fields may be inappropriately instantiated.	22486484, 22431364	100
AvoidThrowingNull PointerException	The use of NullPointerExceptions may cause confusion if it is assumed that an error has been thrown by the virtual machine rather than due to an error in the code.	24458961, 27019889	0
AvoidThrowingRaw ExceptionTypes	By throwing a 'RuntimeException', 'Throwable', 'Exception' or 'Error', the cause of the error might be unclear.	21117967, 24694493, 26122332, 23198269, 23291448, 25600796, 24393070, 25474907	0
ClassWithOnlyPrivate ConstructorsShould BeFinal	A class that is not 'final' and has only private constructors may cause confusion and security breaches.	20940715, 25646266, 22196185, 26044625, 22436954	0

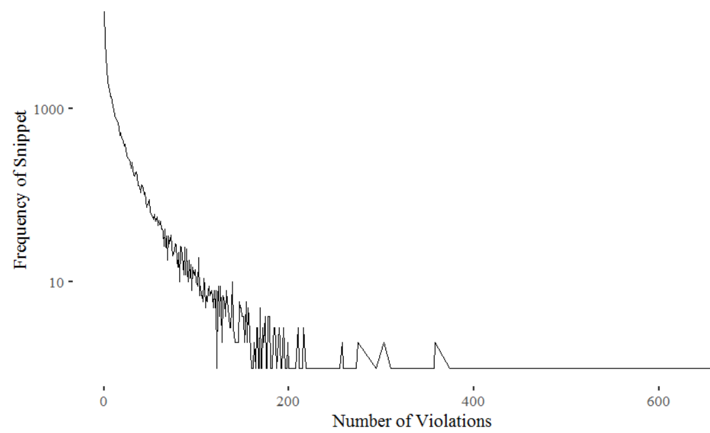
ConstructorCalls OverridableMethod*	A constructor calling an overridable method may cause it to be invoked on an object which has not been entirely constructed or cause debugging challenges.	25467866, 22443386, 26145664, 21142696, 21128671, 26028341, 23943042	0
DoubleChecked Locking	Double-checked locking improves the performance of accessing objects, but it can provide debugging challenges.	23721115	100
EmptyMethodIn AbstractClassShould BeAbstract	A non-'abstract' empty method inside an 'abstract' class may allow the class to be used by developers inappropriately.	21198215, 26662116, 23128335	33.33
ReturnEmptyArray RatherThanNull*	Returning a null reference may cause an unexpected 'NullPointerException', which makes the code more complicated.	27386421, 23568386	0

***Reliability violations**

4.2 Readability (RQ2)

4.2.1 All Readability Violations

Of the 50,717 parsable code snippets coming out of the pipeline in the previous section, 50,472 codes snippets could be analysed by Checkstyle to evaluate readability. As noted above, this reduced number is due to UTF-8 characters being unsupported by Checkstyle. This resulted in 530,521 violations for 50,472 code snippets. Figure 10.a shows the same trend in the number of readability violations and frequency of snippets as for the number of reliability and conformance to programming rules violations and frequency of snippets (see Figure 7.a). However, there are more snippets with a larger number of readability violations than those for reliability and conformance to programming rules violations. Grouped into bins of five, as shown in Figure 10.b, 44.41% (16,604) of code snippets with violations contained between one and five violations, 18.18% (6,797) contain between six and 10 violations, 10.68% (3,995) contain between 11 and 15 violations, 6.79% (2,538) contain between 16 and 20 violations, and 19.94% (7,457) contain more than 20 violations. On average, there were 10.51 violations per code snippet, with the minimum number of violations per code snippet being zero and the maximum number of violations being 667. Table 9 shows that the most common category of violation was *Indentation* with 340,499 (64.18%) violations, followed by *Whitespace* with 104,220 (19.64%) violations, *Blocks* with 29,266 (5.52%) violations and *Javadoc* with 21,186 (3.99%) violations. Others were less frequent in Table 9 (Naming=15,147, Sizes=7,798, Imports=4,384, Coding=3,452, Other=2,272, Design=1,724, Modifier=504 and Annotation=69). We next drill down further into the snippet relevant violations.



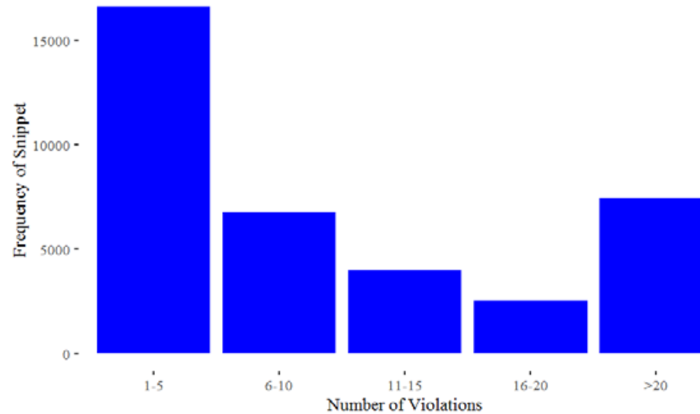


Figure 10. Number of readability violations per snippet (a) and Number of readability violations grouped (b)

Table 9. Number of readability violations grouped by violation category

Rules violated by category	Number of Violations	Percentage of Violations
Indentation	340499	64.18%
Whitespace	104220	19.64%
Blocks	29266	5.52%
Javadoc	21186	3.99%
Naming	15147	2.86%
Sizes	7798	1.47%
Imports	4384	0.83%
Coding	3452	0.65%
Other	2272	0.43%
Design	1724	0.32%
Modifier	504	0.10%
Annotation	69	0.01%
Total	***530521***	

4.2.2 Snippet Relevant Violations

Of the 48 readability violations considered to be relevant to code snippets, the 20% most relevant violations (10 violations) for code snippets were determined. As the Checkstyle tool does not assign a priority to each violation, the violation types were ranked based on a combination of perceived relevance, importance of violation and number of violations (see Table 10). As with the reliability check to decide on the relevance of the violation to code snippets in Section 3.2.5, authors two and three examined these checks agreeing on their relevance over the others. The most relevant violation selected was *LeftCurlyCheck*, which is a frequent violation (18051 violations) that provides inconsistency with Java coding standards. Other violations in Table 10 occurred less frequently; however, these were still common (e.g., *NeedBracesCheck*, *RightCurlyCheck* and *AvoidStarImportCheck*). The least relevant violation included in the top 20% of readability violations was *UpperEllCheck* because, although the lowercase letter ‘l’ can

be confused with the number 1, this problem is relatively infrequent (64 violations) and may not be severely troublesome to developers. We review all of these violations with deeper qualitative analysis in the next subsection.

Table 10. Readability violations most relevant to code snippets

Rule Violated	Number of Violations	Percentage of Violations
LeftCurlyCheck	18051	3.40
NeedBracesCheck	6519	1.23
RightCurlyCheck	4082	0.77
AvoidStarImportCheck	2252	0.42
ArrayTypeStyleCheck	2131	0.40
VariableDeclarationUsageDistanceCheck	1218	0.23
ModifierOrderCheck	504	0.10
MissingSwitchDefaultCheck	278	0.05
AvoidEscapedUnicodeCharactersCheck	77	0.01
UpperEllCheck	64	0.01

4.2.3 Qualitative Analysis

As in Section 4.1.3, we sampled 60 snippets covering all types of readability violation for our qualitative analysis. This sample covered 50 different violations of those 56 in the list for Appendix D.2 (one violation was removed as it was caused by our pre-processing steps and five were not detected in the code snippets). Of the 50 violations, there was no context provided around the snippets (e.g., in question, answers, comments) which may lead to fixing 44 types of violations (88%) or an awareness that these exist (refer to Table 11). The opposite was observed for the remaining six or 12% (*EmptyBlockCheck*, *RightCurlyCheck*, *FallThroughCheck*, *NoFinalizerCheck*, *UpperEllCheck*, *OperatorWrapCheck*).

The use of a statement with an empty code block (*EmptyBlockCheck*) provides confusion about the purpose or importance of the block. On the other hand, the use of an incorrect right bracket (*RightCurlyCheck*) may make the end of a code block (e.g., method, class) less obvious, causing important code to be left out of the block. For a 'switch' statement, a 'case' block that does not end with a 'break' statement (*FallThroughCheck*) is likely to cause confusion about the process flow associated with the code as well as incorrect execution of code statements for a particular state/condition. A 'finalize' method (*NoFinalizerCheck*) may exhibit unpredictable behaviour, which may cause problems in terms of debugging/testing, performance and portability. The use of a lowercase 'l' (*UpperEllCheck*) may cause confusion about the type of a data structure and, therefore, the possible operations associated with it. Finally, while incorrectly positioning operators in a statement over multiple code lines (*OperatorWrapCheck*) is a minor issue, it may cause the code statement to be less readable (particularly for large amounts of code). Code snippets and commentary adhering to these coding rules may be assessed as highly readable, and it was pleasing to see the Stack Overflow community demonstrating such awareness.

For example, in considering the *EmptyBlockCheck* violation for the code snippet associated with Question ID 23392208, the question makes it clear that the purpose of the code snippet is to show how to distinguish between object types 'List<Cat>' and 'List<Orange>'. This is why the code has empty 'if' blocks. However, the purpose of the code would have been clearer if comments had been included in the 'if' blocks (e.g., "//do something"). The answer associated with the code snippet for Question ID 23350956, written on April 28, 2014, only states that the 'switch' statement should start with "case 0" rather than "case 1". However, another answer (written on April 29, 2014) states that a 'break' statement is needed at the end of each 'case' block (refer to Figure 11.a), thus providing context for reducing the likelihood of the *FallThroughCheck* violation if the code was reused.

The evidence for the lack of context for 44 out of 50 violations (88%) is not good for Stack Overflow code readability overall however. Such errors include: *AnnotationLocationCheck*, *ArrayTypeStyleCheck*, *EmptyCatchBlockCheck, NeedBracesCheck*, *MultipleVariableDeclarationsCheck*, *OverloadMethodsDeclarationOrderCheck*, *AvoidStarImportCheck*, *NonEmptyAtclauseDescriptionCheck*, *ModifierOrderCheck*, *ParameterNameCheck* and *SeparatorWrapCheck* (refer to Table 11 for full details). For instance, although not serious, the indentation of annotations (*AnnotationLocationCheck*) makes code less readable. This may also have a flow-on effect to code below the annotations that are also incorrectly indented and therefore harder to read or debug (e.g., which statements/structures are associated with which brackets). This is the case for the code snippet associated with Question ID 27692431 (refer to Figure 11.b), as the class declaration has the same

indentation as the annotation (we grouped two other indentation checks with this violation in Table 11). The use of an empty ‘catch’ block (*EmptyCatchBlockCheck*) may lead to confusion about the purpose or importance of the block (refer to answer for Question ID 24427892). Also, there may be usability and security issues associated with not dealing with an identified error. Multiple variable declarations (*MultipleVariableDeclarationsCheck*) on one line can make the code less readable in terms of finding variable declarations and traversing longer code lines (refer to answer for Question ID 25873979). Importing all classes in a package (*AvoidStarImportCheck*) leads to tight coupling between packages, which may cause errors and confusion when debugging in terms of naming conflicts. This may also increase compilation time (refer to answer for Question ID 27012261). The incorrect order of modifiers (*ModifierOrderCheck*) may cause confusion about the state or visibility of a class or variable and therefore issues associated with security (refer to answer for Question ID 26948858). While incorrectly positioning separators in a statement over multiple code lines (*SeparatorWrapCheck*) is a minor issue, it may cause the code statement to be less readable and lead to confusion about the relationship between the variable and its method (refer to answer for Question ID 25513263).

We provide additional details for 50 readability violations in Table 11. In particular, we list the rule that was violated, the implication for breaking the rule (for code reuse) and degree of context (in %) that is provided around the snippets (e.g., in question, answers, comments) which may lead to fixing the violation or an awareness that it exists.

a. Another Answer Associated with Question ID 23350956

So, first, you will want to modify the `switch` so that the `case`s match the same order as the `getValueAt` method and add a `break` statement after each `case` in order to prevent the following `case`s from been executed, for example...

```
public void setValueAt(Object value, int row, int column)
{
    switch(column)
    {
    case 0:
        basketItems.get(row).setBrand((String)value);
        break;
    case 1:
        basketItems.get(row).setModel((String)value);
        break;
    case 2:
        basketItems.get(row).setPrice((double)value);
        break;
    case 3:
        basketItems.get(row).setQuantity(Integer.parseInt((String) value));
        break;
    case 4:
        basketItems.get(row).setTotalPrice(Double.parseDouble((String) value));
        break;
    }
}
```

b. Code Snippet Associated with Question ID 27692431

```
1  @Configuration
2      @PropertySource(value={
3      "classpath:usermgmt.properties",
4      "classpath:ldap.properties",
5
6       })
7       @ContextConfiguration(locations = {
8      "file:src/main/webapp/WEB-INF/spring-config.xml",
9      "file:src/main/webapp/WEB-INF/conf/applicationContext-email.xml",
10     "file:src/main/webapp/WEB-INF/conf/applicationContext-jdbc.xml"
11
12    })
13    @WebAppConfiguration
14    @RunWith(SpringJUnit4ClassRunner.class)
15    public class ClassTest {
16
17    }
```

Figure 11. Another answer associated with Question ID 23350956 (a) and Code snippet associated with Question ID 27692431 (b)

Table 11. Qualitative analysis for readability violations

Rule	**Implications of Breaking Rule**	**Question IDs**	**Context Would Fix Error (%)**
AnnotationLocationCheck, CommentsIndentationCheck, IndentationCheck	Inappropriate indentation affects code readability and could lead to confusion when debugging.	27692431	0
ArrayTypeStyleCheck	The use of incorrect coding style for an array type may lead to accidental misuse.	25184589	0
AvoidEscapedUnicodeCharacters Check	The use of a Unicode escape character makes it unclear what is being represented, which makes the code hard to understand.	24177348	0
EmptyBlockCheck	The use of a statement with an empty code block may lead to confusion about the purpose or importance of the block.	23392208	100
EmptyCatchBlockCheck	The use of an empty ‘catch’ block may lead to confusion about the purpose of the block and usability and security issues.	24427892	0
LeftCurlyCheck	The use of an incorrect left bracket style makes it less clear that the code is written in Java and encourages accidentally use.	23041258, 26649619, 27867492	0
NeedBracesCheck	Not including brackets around a code block may cause confusion about the process flow associated with the code.	27606577	0

RightCurlyCheck	The use of an incorrect right bracket style at the end of a code block (e.g. method) causes important code to be left out.	24361899	100
FallThroughCheck	For a ‘switch’ statement, a ‘case’ block that does not end with a ‘break’ statement may cause confusion about the process flow associated with the code and incorrect execution of code.	23350956	100
IllegalTokenTextCheck	The use of an octal or Unicode escape character makes it less clear what is being represented and affects understandability.	22545603	0
MissingSwitchDefaultCheck	By not including a ‘default’ clause, the ‘switch’ statement is unlikely to represent all current possible states/conditions as well as new states introduced in the future, leading to runtime errors.	23771902	0
MultipleVariableDeclarations Check	Multiple variable declarations on one line can make the code less readable in terms of finding variable declarations.	25873979	0
NoFinalizerCheck	A ‘finalize’ method can exhibit unpredictable behaviour, which may cause debugging/testing, performance and portability issues.	23762150	100
OneStatementPerLineCheck	Multiple statements on one line can make the code less readable and wrongly suggest that they are related in terms of code logic.	24619436	0
OverloadMethodsDeclarationOrder Check	By not grouping overloaded method declarations together, it would take longer to find related methods in the code.	23531704	0
VariableDeclarationUsageDistance Check	By having many code lines between a variable declaration and the first statement that uses the variable makes code unreadable.	23789340	0
OneTopLevelClassCheck	By not including a top-level class in its own source file, the file will be larger and unreadable, affecting debugging.	23614212	0
AvoidStarImportCheck	Importing all classes in a package leads to tight coupling between packages, which may cause errors and increase compilation time.	27012261	0
CustomImportOrderCheck	If the import statements are not in the order that the packages are being used the process flow of code is unclear during coding.	26204433	0
AtclauseOrderCheck	Javadoc tags in the wrong order may cause confusion for new developers trying to understand the purpose of a method or class.	27817940	0
JavadocMethodCheck	A method without a Javadoc comment may lead to conflicting interpretations about the purpose of the method and inconsistent code.	23720050, 23960536, 24714712	0
JavadocParagraphCheck	Incorrect usage of paragraph tags may cause Javadoc content to be less readable and lower understandability of classes/methods.	21242110	0
JavadocTagContinuation IndentationCheck	Incorrect indentation of tags may cause Javadoc content to be less readable and hinder understanding of classes/methods.	27796029	0
NonEmptyAtclauseDescription Check	Empty Javadoc tags may cause developers to have inconsistent or conflicting interpretations about the purpose of classes/methods.	22493512	0
SingleLineJavadocCheck, SummaryJavadocCheck	Fragmented summary or incorrect layout of Javadoc comment may cause it to be unreadable and hinder understandability.	26030062, 25421365	0
ModifierOrderCheck	Incorrect order of modifiers may cause confusion about the state or visibility of a class or variable and security issues.	26948858	0
AbbreviationAsWordInNameCheck , CatchParameterNameCheck, ClassTypeParameterNameCheck, InterfaceTypeParameterName Check, LocalVariableNameCheck, MemberNameCheck, MethodNameCheck, MethodTypeParameterNameCheck, PackageNameCheck, ParameterNameCheck, TypeNameCheck	By not following Java naming style standards, it less clear to developers what type of structure is being represented when code references a particular name. This may hinder understanding of the purpose of the code and cause confusion about the possible operations associated with a structure.	25934321, 21202523, 22856547, 21908768, 27184879, 26386057, 26752021, 23061313, 27068654, 23774985, 21238083	0
LineLengthCheck	Use of long lines of code makes the file less readable and may require horizontal scrolling, slowing down programming time.	26450282	0
UpperEllCheck	The use of a lowercase ‘l’ may cause confusion about the type of a data structure and the operations associated with it.	27336026	100
EmptyLineSeparatorCheck	Declarations that are not separated by empty lines are less readable and confuses the start, end and scope of a structure.	26228495	0
FileTabCharacterCheck	Use of tab characters in code may require developers to configure the tab width of their text editor, and increase readability issues.	26945024	0
GenericWhitespaceCheck	Whitespace around a generic type physically separates a variable type from its generic type leading to understandability issues.	23448238	0
MethodParamPadCheck	Whitespace around method parameter brackets physically separates a method name call from its parameters, causing developers to think that there is code missing from the statement.	22389102	0
OperatorWrapCheck	Incorrectly positioning operators in a statement over multiple code lines may cause the code statement to be less readable.	26304185	100
ParenPadCheck	Whitespace around method parameters may confuse the relationship between the method name and the parameters.	26630799, 25644671	0

SeparatorWrapCheck	Incorrectly positioning separators in a statement over multiple code lines may cause the code statement to be less readable.	25513263	0
WhitespaceAroundCheck	While a lack of whitespace around operators or brackets is a minor issue, it can cause the code statement to be less elegant or less readable. This may hinder understanding of the code.	27306448, 22688682, 22263989, 25825289, 25276828, 26491038, 25106112, 27369075	0

4.3 Performance (RQ3)

4.3.1 All Performance Violations

Of the 50,472 code snippets analysed by Checkstyle that were then compiled, 8,010 code snippets could be analysed by FindBugs to evaluate performance and security. Of the results in the performance category there were 3,949 violations for 2,936 (36.65%) code snippets. This means that 5,074 (63.35%) code snippets had no performance violations. Figure 12.a shows the same trend in the number of performance violations as for reliability and conformance to programming rules and readability violations (see Figures 7.a and 10.a). However, there are less snippets with a small number of violations because fewer snippets could be analysed by FindBugs than PMD and Checkstyle and there are fewer performance violation checks than for reliability and conformance to programming rules or readability. As shown in Figure 12.b, of the code snippets that had violations, 2,719 (92.61%) code snippets contain between one and two violations, 182 (6.20%) code snippets contain between three and four violations, 28 (0.95%) code snippets contain between five and six violations, four code snippets (0.14%) contain between seven and 10, and only three code snippets (0.10%) contain more than 10 violations. The minimum number of performance violations in a single code snippet was zero, while the maximum was 18, with an average of 0.49 violations per snippet, across all snippets assessed. The majority of the performance violations were from the *Unread field* category (63.51%, or 2,508), followed by *Unused field* (15.37%, 607) and *Private method is never called* (6.89%, 272). These outcomes are shown in Table 12.

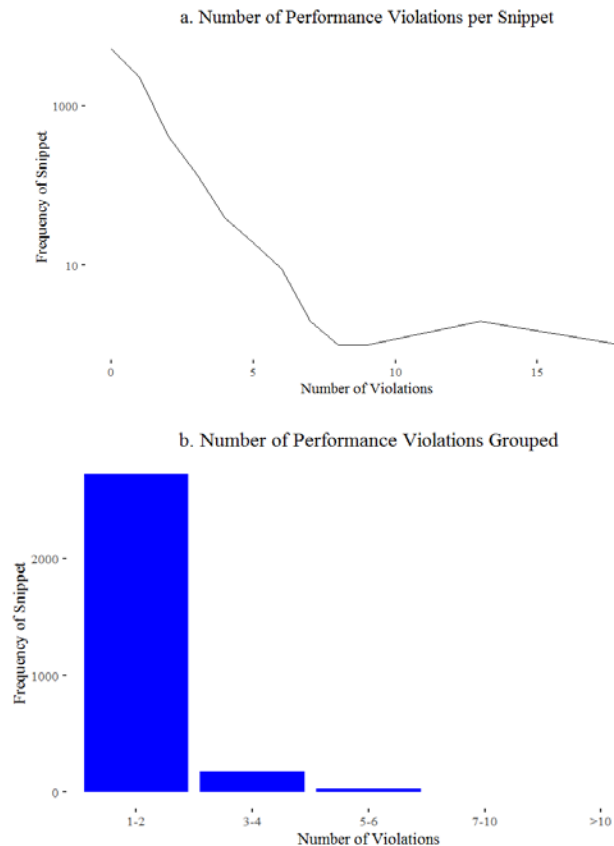


Figure 12. Number of performance violations per snippet (a) and Number of performance violations grouped (b)

Table 12. Number of performance violations grouped by bug category

Rules violated by category	Number of Violations	Percentage of Violations
Unread field	2508	63.51%
Unused filed	607	15.37%
Private method is never called	272	6.89%
Inner class could be static	173	4.38%
Dubious method used	132	3.34%
String concatenation in loop	90	2.28%
Questionable boxing of primitive value	84	2.13%
Unread field should be static	73	1.85%
Inefficient map iterator	7	0.18%
Unnecessary Math on constants	3	0.08%
Total	***3949***	

4.3.2 Snippet Relevant Violations

Of the 18 performance violations considered to be relevant to code snippets, the 20% most relevant violations (five violations) for code snippets were determined. As the FindBugs tool assigns a priority to each violation, the violation types were ranked in order of priority (starting with the highest priority violations) followed by number of violations (see Table 13). The most relevant violation was *Dm_Boxed_Primitive_For_Parsing*, which is associated with a high priority because parsing a value is a common programming operation and, therefore, should be performed in a computationally efficient way. Although *Dm_Boxed_Primitive_Tostring* is associated with a greater number of violations compared to *Dm_Gc*, *Dm_Boxed_Primitive_Tostring* is associated with a greater number of priority one violations (most critical). The least relevant violation included in the top 20% of violations was *Sic_Inner_Should_Be_Static* because although there were a greater number of these violations (88 violations in Table 13) compared to *Dm_Boxed_Primitive_For_Parsing*, based on the priority values, the computational efficiency of inner classes is considered less important than the computational efficiency of operations on a single variable.

Table 13. Performance violations most relevant to code snippets

Rule Violated	Priority	Number of Violations	Percentage of Violations
Dm_Boxed_Primitive_For_Parsing	1	14	2.49
Dm_Gc	1	2	0.36
Dm_Boxed_Primitive_Tostring	1,2	5	0.89
Sbsc_Use_Stringbuffer_Concatenation	2	90	16.01
Sic_Inner_Should_Be_Static	2	88	15.66

4.3.3 Qualitative Analysis

For the qualitative analysis, our sample of 60 code snippets covered 21 performance violations (refer to D.3 for full coverage of performance checks). In Table 14, 14 of the 21 performance violations (66.7%) would be overcome by the Stack Overflow community as there was context provided around the snippets (e.g., in question, answers, comments) which may lead to fixing these violations or an awareness that these exist (refer to Table 14). There was a lack of awareness for the remaining 7 violations (or 33.3%).

These neglected performance violations include: *Bx_Boxing_Immediately_Unboxed_To_Perform_Coercion*, *Dm_Nextint_Via_Nextdouble*, *Dm_Number_Ctor, Sic_Inner_Should_Be_Static*, *Sic_Inner_Should_Be_Static_Anon*, *Sic_Inner_Should_Be_Static_Needs_This*, *Wmi_Wrong_Map_Iterator* (refer to Table 14 for details). These violations negatively affect software performance for many reasons. For instance, boxing and then immediately unboxing a primitive value (*Bx_Boxing_Immediately_Unboxed_To_Perform_Coercion*) in order to convert the type of the value unnecessarily increases compilation time. Under conditions where it is necessary to optimise compilation time, such violations would be an impediment. In the same way, generating a floating-point value and then converting it to an integer involves an unnecessary conversion (*Dm_Nextint_Via_Nextdouble*), which may slow down code and affect its readability. The use of the *number* constructor always creates a new object and, therefore, does not allow the caching of values (*Dm_Number_Ctor*). This uses unnecessary memory and slows down code. Furthermore, the inner class's embedded reference to the outer class object makes the instances of the inner class larger and may keep the reference available longer than necessary, which wastes memory (*Sic_Inner_Should_Be_Static*, *Sic_Inner_Should_Be_Static_Anon*, *Sic_Inner_Should_Be_Static_Needs_This*).

Finally, the use of the 'keySet' iterator method is inefficient and therefore may slow down the program (*Wmi_Wrong_Map_Iterator*). These are undesirable performance issues in Stack Overflow code snippets for which the community seem to lack awareness or there is oversight.

On the contrary, contributors seem aware of other performance violations, and fittingly suggested interventions to overcome these (see Table 14). For instance, boxing and then immediately unboxing a primitive value forces the compiler to undo the work of the boxing (*Bx_Boxing_Immediately_Unboxed*), which increases compilation time. We observed that for the code snippet in response to Question ID 21597991 (refer to Figure 13.a), it was clear from the comments on code lines 3 and 4 that these lines should be removed when retrieving input from the command line (code lines 5 and 6). This issue is also discussed by the contributor in the text that surrounded the snippet. The use of the floating-point number constructor always creates a new object and, therefore, does not allow the caching of values (*Dm_Fp_Number_Ctor*). This uses unnecessary memory and slows down code. We observed that for the code snippet in response to Question ID 26260792, it was clear from the third comment posted below the answer that "amount.intValue()" and "amount.doubleValue()" could be used "instead of casting". If the question author or a developer copied the code snippet for reuse, they are likely to edit the snippet based on the comment. The unnecessary use of a 'Math' class method may slow down the program and make the code less readable (*Um_Unnecessary_Math*). We observed that for the code snippet in response to Question ID 21510763, it was clear that the contributor's goal was to provide details for the community to understand the behaviour of the 'Math.ceil' method (refer to Figure 13.b). This is illustrated through the use of constant values in the answer code snippet.

While there were numerous performance violations in Stack Overflow code snippets, our deeper analysis shows that the community was more aware of these violations than the reliability and conformance to programming rules violations and readability violations (reliability and conformance to programming rules=37.5%, readability=12%, performance=66.7%).

a. Code Snippet Associated with Question ID 21597991

```
1  public class C42423{
2  public static void main(String[] args){
3     int number = new Integer(5); // you can comment this line when providing input from command line
4     String word = new String("hello");// you can comment this line when providing input from command line
5     number = Integer.parseInt(args[0]);
6     word = args[1];
7
8     String method = method1(word,number);
9     System.out.println(method);
10 }
11
12 public static String method1(String word, int number) {
13    if (number == 0){
14       return "";
15    }
16    else{
17       return word + method1(word,number-1);
18    }
19 }
20
21 }
```

b. Answer Associated with Question ID 21510763

This is Integer division. In Java:

(30 * 50) / 1000 = 1

```
int i = (int) Math.ceil((30.0 * 50) / 1000);
```

Will give you the expected result.

Figure 13. Code snippet associated with Question ID 21597991 (a) and Answer associated with Question ID 21510763 (b)

Table 14. Qualitative analysis for performance violations

Rule	**Implications of Breaking Rule**	**Question IDs**	**Context Would Fix Error (%)**
Bx_Boxing_Immediately_ Unboxed	Boxing and then immediately unboxing a primitive value increases compilation time.	21597991	100
Bx_Boxing_Immediately_ Unboxed_To_Perform_ Coercion	Boxing and then immediately unboxing a primitive value in order to convert the type of the value unnecessarily increases compilation time.	26807872	0
Dm_Boxed_Primitive_For_ Parsing	Boxing and then unboxing a primitive value for parsing is inefficient, increasing compilation time.	23014428	100
Dm_Boxed_Primitive_Tostring	The use of a boxed primitive only for calling 'toString()' creates a new object and slows down code.	27096670	100
Dm_Fp_Number_Ctor	The use of the floating-point number constructor always creates a new object; this does not allow the caching of values and slows down code.	26260792	100
Dm_Gc	Explicit garbage collection is computationally expensive and can slow down a program.	23831157	100

Dm_Nextint_Via_Nextdouble	Generating a random floating-point value and then converting it to an integer involves an unnecessary conversion, which may slow down (unreadable) code.	25926194, 26734389	0
Dm_Number_Ctor	The use of integer number constructors always creates a new object, slowing down code.	25150130, 23273936	0
Dm_String_Ctor	The use of the String constructor wastes memory and increases the length of code lines.	22082451, 27417861	100
Dm_String_Tostring	Invoking a String's 'toString()' method makes the code less readable and may slow down the program.	21329215	100
Dm_String_Void_Ctor	The use of the String constructor with no arguments wastes memory, slows down the program and increases the length of code lines.	20926110	100
Sbsc_Use_Stringbuffer_ Concatenation	Concatenating a string using the '+' operator in a loop is inefficient and slows down the program.	26873488, 21589139	100
Sic_Inner_Should_Be_Static, Sic_Inner_Should_Be_Static_ Anon, Sic_Inner_Should_Be_Static_ Needs_This	The inner class's embedded reference to the outer class object makes the instances of the inner class larger and may keep the reference available longer than necessary, which wastes memory.	22059165, 21010495, 24213048, 26559724, 25396727	0 0 0
Ss_Should_Be_Static	The use of an instance/non-static variable that has the same value across instances of a class wastes memory.	26999888, 22662566	50
Um_Unnecessary_Math	The unnecessary use of a 'Math' class method may slow down the program and make the code less readable.	21510763	100
Upm_Uncalled_Private_ Method	An uncalled private method makes a code file longer, slows down compilation time and may be a source of confusion.	26093739, 27851789, 26269209, 24478027	100
Urf_Unread_Field	An unread field wastes memory and multiple unread fields would make the Java class file longer. It may also lead to confusion about the purpose of the field and related code. However, for a substantial software development project, there may be a lag between declaring/initialising a field and the implementation of functionality that uses the field.	27087229, 22567272, 27741462, 24066627, 25336361, 20973988, 24780956, 22772664, 21531089, 22049133, 24098269, 21608283, 22680424, 22044902, 24432977, 24961245, 21950439, 22521144, 22758214, 24372932, 24651403, 22316800	100
Uuf_Unused_Field	An unused field may cause an error when compiling as well as errors due to the default value of the field. It may also lead to confusion about the purpose of the field and related code.	21204589, 24865229, 23727501 (x2), 21972809, 22541952, 26936567, 22138665, 26262857	88.89
Wmi_Wrong_Map_Iterator	The use of the 'keySet' iterator method is inefficient and may slow down the program, particularly for many iterations.	25796079	0

4.4 Security (RQ4)

4.4.1 All Security Violations

The same 8,010 code snippets that were examined to analyse performance were also used to analyse security, as both aspects were analysed using the FindBugs tool. Of the results in the security and malicious code vulnerability categories, there were 75 FindBugs violations for 59 (0.74%) code snippets. This means that 7,951 (99.26%) of code snippets did not contain any violations relating to security. Figure 14 shows that of the code snippets that had violations, 46 (77.97%) code snippets contained one violation, 11 (18.64%) code snippets contained two violations, with a single code snippet containing three and four violations. On average, there were 0.01 violations per snippet, with the minimum number of security violations in a single code snippet being zero, and a maximum of four, across all snippets assessed.

Table 15 shows that a majority of the security violations were in the *mutable static field* category, 64/75 (85.33%), which relates to the visibility and types of variables. In addition, *use doPrivileged* had five violations (6.67%), *expose internal representation* had three violations (4.00%), *dubious method used*[20] had two violations (2.67%), and *string reference to mutable object* had a single violation (1.33%).

[20] While 'dubious method used' is a type of bug in both the performance and security categories, they represent different bugs.

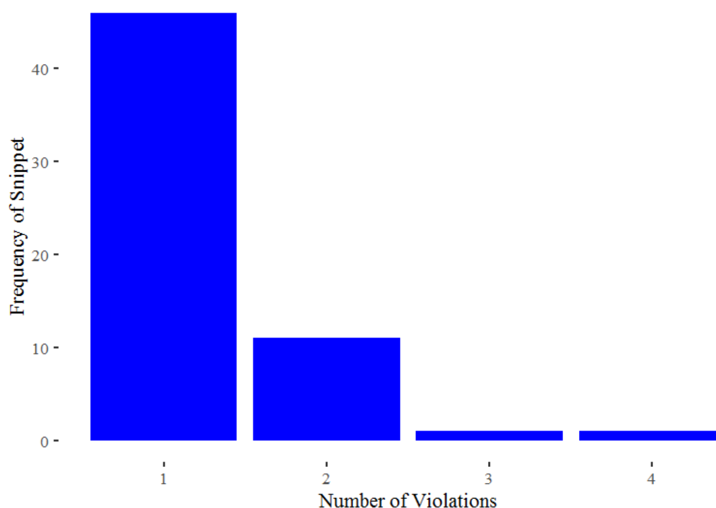


Figure 14. Number of security violations per snippet

Table 15. Number of security violations grouped by bug category

Rules violated by category	Number of Violations	Percentage of Violations
Mutable static field	64	85.33%
Use doPrivileged	5	6.67%
Expose internal representation	3	4.00%
Dubious method used	2	2.67%
Storing reference to mutable object	1	1.33%
Total	***75***	

4.4.2 Snippet Relevant Violations

Of the 11 security violations considered to be relevant to code snippets, the 20% most relevant violations (three violations) for code snippets were determined. As the FindBugs tool assigns a priority to each violation, the violation types were ranked in order of priority (starting with the highest priority violations) followed by number of violations (see Table 16). While both *Ei_Expose_Rep* and *Dp_Create_Classloader_Inside_Do_Privileged* exhibit two violations associated with priority two, *Ei_Expose_Rep* is considered to be more important because passing mutable object references between different components of a system is a common operation and, therefore, should not be vulnerable to security breaches. The least relevant violation included in the top 20% of violations was *Ms_Final_Pkgprotect* because although there were a greater number of these violations (6 violations) compared to *Dp_Create_Classloader_Inside_Do_Privileged* or *Ei_Expose_Rep*, *Ms_Final_Pkgprotect* is associated with fewer priority two violations. We examine these violations in greater depth in the next subsection.

Table 16. Security violations most relevant to code snippets

Rule Violated	Priority	Number of Violations	Percentage of Violations
Ei_Expose_Rep	2	3	4.00
Dp_Create_Classloader_Inside_Do_Privileged	2	3	4.00
Ms_Final_Pkgprotect	2,3	6	8.00

4.4.3 Qualitative Analysis

Finally, in following the same process as was used for our deeper qualitative analysis of reliability and conformance to programming rules, readability and performance violations, we sampled 60 violations from the available 59 code snippets that returned security violations for qualitative analysis. We sampled all categories of security violations, exploring the rule that was violated, the implication for breaking the rule (for code reuse) and context that is provided around the snippets (e.g., in question, answers, comments) which may lead to fixing the violation or an awareness that it exists. Altogether, Stack Overflow code snippets violated 11 security checks (refer to Table 17). Evidence in Table 17 shows that violations for four of these checks (36.4%) would be overcome by the Stack Overflow community as there was context provided around the snippets (e.g., in question, answers, comments)

which may lead to fixing many of these violations or an awareness that they exist. There was less awareness around the other seven violations (63.4%) however, which were missed by the Stack Overflow community.

The Stack Overflow community were not aware of (or did not worked to address) the *Dmi_Constant_Db_Password*, *Dp_Create_Classloader_Inside_Do_Privileged*, *Dp_Do_Inside_Do_Privileged*, *Ei_Expose_Rep2*, *Ei_Expose_Static_Rep2, Ms_Expose_Rep* and *Ms_Pkgprotect* violations. A hardcoded database password (*Dmi_Constant_Db_Password*) allows anyone with access to the source code or compiled code to learn the password. This is likely to cause a security breach. If a security manager is installed, creating a classloader requires permissions (*Dp_Create_Classloader_Inside_Do_Privileged*) and, therefore, may require a 'doPrivileged' block (temporarily providing greater privileges in order to perform the task) in order to avoid errors. If a security manager is not installed, creating a classloader may cause a security breach. A similar threat persists for the *Dp_Do_Inside_Do_Privileged* violation. If a security manager is installed, calling a method that requires permissions may need a 'doPrivileged' block (temporarily providing greater privileges in order to perform the task) in order to avoid errors. Storing a reference to an externally mutable object in a (static) field may cause a security breach; the object may be involved in unchecked changes and accessed by untrusted code (*Ei_Expose_Rep2* and *Ei_Expose_Static_Rep2*). A 'public' and 'static' method that returns a reference to a 'static' array may cause a security breach because any code that calls the method can modify the array (*Ms_Expose_Rep*). Finally, a mutable 'static' field that is 'public' could be changed by malicious code, causing a security breach (*Ms_Pkgprotect*). These are undesirable security issues in Stack Overflow code snippets for which the community seem to lack awareness or there is oversight.

More positive, the Stack Overflow community mitigated *Ei_Expose_Rep*, *Ms_Final_Pkgprotect*, *Ms_Ooi_Pkgprotect* and *Ms_Should_Be_Final* violations. Returning a reference to a mutable object field value exposes the internal representation of the object associated with the field (*Ei_Expose_Rep*). Unchecked changes to the mutable object may cause a security breach. For the code snippet in response to Question ID 27867068 (refer to Figure 15.a), another answer includes a *get* method that returns a clone of the array. However, this answer is at the bottom of the page, which may lead to contributors missing it. A mutable 'static' field that is not 'final' or 'package' protected could be accidentally changed by code in another package (*Ms_Final_Pkgprotect*). Also, the field could be changed by malicious code, causing a security breach. For the code snippet in response to Question ID 21668288, another answer uses a 'private' field. However, this answer is at the bottom of the page; notwithstanding also that the answer associated with the violation has been accepted by the question author. An interface field that is 'public' and references a mutable object could be accidentally changed by code in another package (*Ms_Ooi_Pkgprotect*). Also, the field could be changed by malicious code, causing a security breach. For the code snippet in response to Question ID 21936973 (refer to Figure 15.b), one of the comments posted below the question advises "against declaring variables in interfaces". However, there is no discussion about removing the 'public' modifier. Also, the comment is not visible without clicking the 'show more comments' button. Therefore, the question author and Stack Overflow readers may not see the comment. Finally, a 'public' and 'static' field that is not 'final' could be changed by malicious code (*Ms_Should_Be_Final*), causing a security breach. Also, the field could be accidentally changed by code in another package. For the code snippet in response to Question ID 25305523, the purpose of the answer is to compare *enums* and *static constants* in terms of memory usage rather than to solve a problem relating to a specific program. Therefore, community members may become aware of this when the snippet is reused, avoiding the error in the process. Altogether, there was mixed evidence around Stack Overflow contributors' vigilance of the reliability and conformance to programming rules, readability, performance and security errors that were evident in code snippets.

a. Another Answer Associated with Question ID 27867068

Not in the sense you're thinking, but this can be accomplished more straightforwardly.

If those enums are capable of holding an array of `States`, you can set their known states at declaration time.

Example:

```
public enum Modules {
    ATTACK(States.IDLE, States.NEAREST, States.NEARESTTOHQ),
    MOVE(States.IDLE, States.NORTH, States.SOUTH, States.EAST, States.WEST),
    SPAWN(States.IDLE, States.SIMPLESPAWN);

    private final States[] knownStates;

    private Modules(States... knownStates) {
        this.knownStates = knownStates;
    }

    public States[] getKnownStates() {
        return knownStates.clone();
    }
}
```

b. Code Snippet and Comments Associated with Question ID 21936973

```
1 public class C62037{
2 public interface Trail {
3     public static final String[] STRINGS = {"bla","bla","bla","bla", "bla", "bla"};
4 }
5
6 }
```

2 Can we see your complete code, pls ? – Feb 21 '14 at 14:22

"can you only work with integer constants in an interface?" No. – Feb 21 '14 at 14:22

Where is Test class code – Feb 21 '14 at 14:23

The class is 500 lines long but I will get the relevant lines now – Feb 21 '14 at 14:23

Hie, I just tested it here, it work as expected, i can access the static strings as if it was declare in implementing class. The problem is elsewhere – Feb 21 '14 at 14:26

3 Note that this isn't really a constant array. Anyone can do `Constants.STRINGS[1] = null;` . To guarantee immutability, you should use `List<String> STRINGS = Collections.unmodifiableList(Arrays.asList("bla", "bla"));` – Feb 21 '14 at 14:26

Thanks for the tip JB. Edits made to the question guys – Feb 21 '14 at 14:27

Even though it is possible, I would advise *against* declaring variables in interfaces. You can create custom classes for that. – Feb 21 '14 at 14:27

Figure 15. Another answer associated with Question ID 27867068 (a) and Code snippet and comments associated with Question ID 21936973 (b)

Table 17. Qualitative analysis for security violations

Rule	**Implications of Breaking Rule**	**Question IDs**	**Context Would Fix Error**
Dmi_Constant_Db_ Password	A hardcoded constant database password allows anyone with access to the source code or compiled code to learn the password. This is likely to cause a security breach.	21361781, 25453794	0
Dp_Create_Classloader_ Inside_Do_Privileged	If a security manager is installed, creating a classloader requires permissions and therefore may require a 'doPrivileged' block (temporarily providing greater privileges in order to perform the task) in order to avoid errors.	25897954	0
Dp_Do_Inside_Do_ Privileged	If a security manager is installed, calling a method that requires permissions may need a 'doPrivileged' block (temporarily providing greater privileges in order to perform the task) in order to avoid errors.	23119851, 25869837	0
Ei_Expose_Rep	Returning a reference to a mutable object field value exposes the internal representation of the object associated with the field. This may cause a security breach.	27867068, 22200047, 23855548	33.33
Ei_Expose_Rep2	Storing a reference to an externally mutable object in a field may cause a security breach; the object may be involved in unchecked changes and may be accessed by untrusted code.	23855548	0
Ei_Expose_Static_Rep2	Storing a reference to an externally mutable object in a 'static' field may cause a security breach.	22166799	0
Ms_Expose_Rep	A 'public' and 'static' method that returns a reference to a 'static' array may cause a security breach because any code that calls the method can modify the array.	21668288	0
Ms_Final_Pkgprotect	A mutable 'static' field that is not 'final' or 'package' protected could be accidentally changed by code in another package or malicious code.	25399642, 21407966 (x2), 27305650, 21668288, 23602309	16.67
Ms_Ooi_Pkgprotect	An interface field that is 'public' and references a mutable object could be accidentally or maliciously changed.	21936973	100
Ms_Pkgprotect	A mutable 'static' field that is 'public' could be changed by malicious code, causing a security breach.	26088277, 27670892, 21081297	0
Ms_Should_Be_Final	A 'public' and 'static' field that is not 'final' could be changed by malicious code, causing a security breach. Also, the field could be accidentally changed by code in another package.	21387240, 22963154, 22285223, 26746164, 25382192, 27700614, 25331328, 27659172, 25550225, 22644734, 20942861, 23258715, 25305523, 21909943, 24831177, 25980489, 21591563, 24748098, 21875435, 25705148, 22644734, 26867697, 24635388, 25712918, 26332527, 26714126, 22618624, 23913096, 4335378, 25498133,	48.72

		25236546, 27177079, 27478837, 24450300, 20962911, 21666998, 27670892, 27673848, 24454439	

4.5 Results Summary (Triangulation)

Outcomes in this work show that a majority of reliability and conformance to programming rules violations (88.32%) are priority three ("change recommended"), while only 0.38% are priority one violations ("change absolutely required"). Approximately a third of code snippets assessed had between one and five readability violations, with performance and security code snippets having on average less than one violation. These findings suggest that the quality of Stack Overflow code snippets varies, depending on the dimension being examined. To triangulate these results, Spearman's rank correlation (*rho*) testing was performed for the 8,010 code snippets that were analysed for all four of the quality dimensions (reliability and conformance to programming rules, readability, performance and security). These were assessed considering Cohen's classification, in terms of low ($0 < rho \leq 0.29$), medium ($0.3 \leq rho \leq 0.49$), and high ($rho \geq 0.50$) correlations [18]. We provide correlation results in Table 18, for Stack Overflow code reliability and conformance to programming rules, readability, performance and security. We focus mainly on the medium to high statistically significant correlations given the established view that such results are often noteworthy [18].

The outcomes in Table 18 show that the higher the *Questions Score* received, the more *View Count* such questions received (*rho*=0.33). This pattern of outcome was also similar for *Question Score* and *Answer Score*, with higher scoring questions also attracting higher scoring answers, but with a lower statistically significant correlation (*rho*=0.28). *Code Length* naturally had a strong statistically significant correlation with *Code Spaces* and *LOC*, where these variables were almost linear (*rho*=0.96 for *Code Spaces*, *rho*=0.94 for *LOC*). *Code Spaces* was also strongly correlated with *LOC* (*rho*=0.95). *SPA* had a statistically significant positive correlation with *Answer Score* (*rho*=0.28), which is likely to be due to a more informative answer including more code snippets. However, *SPA* had a statistically significant negative correlation with measures relating to the size of a code snippet (e.g., *rho*=-0.25 for *LOC*). This suggests that an answer that includes multiple code snippets uses each code snippet to focus on a specific operation or piece of functionality, making the answer shorter.

Both the number of reliability and conformance to programming rules violations and the number of readability violations had strong statistically significant correlations with *Code Length*, *Code Spaces* and *LOC* (e.g., *rho*=0.83 for the number of reliability and conformance to programming rules violations and *LOC*). This shows that code that was longer in length tended to have more reliability and conformance to programming rules violations and readability violations. In fact, Table 18 shows that longer code snippets were prone to both types of violations, as we observed that the number of reliability and conformance to programming rules violations had a strong statistically significant correlation with the number of readability violations (*rho*=0.73). In contrast, the number of performance violations and the number of security violations were not strongly correlated with *Code Length*, *Code Spaces* and *LOC* (e.g., *rho*=0.09 for the number of performance violations and *LOC*). Also, the number of performance violations was not correlated with the number of security violations (*rho*=-0.02). In fact, although there was statistically significant correlations between the range of code snippet attributes (measures) and performance and security violations in Table 18, these correlations were all low ($0 < rho \leq 0.29$). Similar outcomes were also observed when performing correlation analysis for code snippets with SPA = 1 separately from those where SPA > 1 (e.g., for the correlation between reliability and conformance to programming rules violations and readability violations, *rho*=0.72 for SPA=1 while *rho*=0.70 for SPA>1).

To probe these outcomes further, correlation testing was also performed for each of the quality dimensions separately, using all of the code snippets analysed by each tool (PMD, Checkstyle and FindBugs). The correlation patterns for each quality dimension were consistent (and almost identical) with the patterns found for the 8,010 code snippets analysed for all of the quality dimensions. The measures relating to the size of a code snippet were strongly correlated with each other across all of the quality dimensions (e.g., *rho*=0.94 for the correlation between *LOC* and *Code Length* for all of the quality dimensions). *Questions Score* had a statistically significant medium correlation with *View Count* for both reliability and conformance to programming rules violations and readability violations (*rho*=0.40) as well as for performance and security violations (*rho*=0.33). There was also a lower correlation

between *Question Score* and *Answer Score* for performance and security violations (*rho*=0.28) but a slightly larger correlation for reliability and conformance to programming rules violations and readability violations (*rho*=0.32). Similar to the correlations shown in Table 18, the number of violations had strong statistically significant correlations with *Code Length*, *Code Spaces* and *LOC* for both reliability and conformance to programming rules violations and readability violations (e.g., *rho*=0.84 for the correlation between number of violations and *Code Length* for reliability and conformance to programming rules violations).

Finally, in considering our correlation analysis for code snippets in accepted and unaccepted answers, it is noted in Table 18 that posts with accepted answers had lower *Answer Count* (*rho*=-0.27) and accepted answers unsurprisingly had both a higher *Answer Score* (*rho*=0.34) and more *SPA* (*rho*=0.24). However, overall, there was no difference in quality between accepted and unaccepted answers. This pattern of outcome was consistent for all quality dimensions that were investigated in the work (reliability and conformance to programming rules, readability, performance and security).

Table 18. Summary code quality correlations

	Measure	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Code Attributes	1. Question Score	1	**0.33*	*0.14	*-0.03	*0.28	*0.03	*0.03	*0.03	*0.08	*-0.03	0.00	0.00	0.01	0.00
	2. View Count		1	*0.20	*0.10	*0.12	*0.15	*0.15	*0.14	0.00	*-0.06	*0.12	*0.12	*0.03	-0.02
	3. Answer Count			1	*0.11	*-0.05	0.01	*0.03	*0.02	*-0.09	*-0.27	*0.04	*0.03	0.00	-0.02
	4. Comment Count				1	0.00	*0.07	*0.09	*0.09	0.00	-0.02	*0.08	*0.08	0.01	0.00
	5. Answer Score					1	*-0.10	*-0.09	*-0.09	*0.28	**0.34*	*-0.09	*-0.13	*0.03	-0.01
	6. Code Length						1	***0.96**	***0.94**	*-0.27	0.00	***0.83**	***0.79**	*0.10	*-0.03
	7. Code Spaces							1	***0.95**	*-0.26	0.00	***0.82**	***0.82**	*0.10	*-0.04
	8. LOC								1	*-0.25	0.01	***0.83**	***0.82**	*0.09	*-0.04
	9. SPA									1	*0.24	*-0.25	*-0.27	0.00	0.02
	10. Accepted (1) /Unaccepted (0)										1	-0.01	-0.01	0.02	-0.01
Number of Violations	11. Reliability and Conformance to Programming Rules (RQ1)											**1**	***0.73**	*0.16	*-0.04
	12. Readability (RQ2)												1	*0.11	*-0.04
	13. Performance (RQ3)													1	-0.02
	14. Security (RQ4)														1

Keys: * indicates significance ($p < 0.05$), *italics* indicates medium effect size using Cohen's classification ($0.3 \leq rho \leq 0.49$), **bold** indicates high effect size using Cohen's classification ($rho \geq 0.50$)

5. DISCUSSION AND IMPLICATIONS

Q&A portals such as Stack Overflow are now central to the way software developers address their knowledge deficiencies when creating software [26]. Thus, there is increasing interest and drive to understand the quality of the content that is provided on such portals [25]. We set out to understand the quality of code snippets that are often provided in Stack Overflow posts. In order to do so, we operationalise code snippet quality along the dimensions of *reliability* and *conformance to programming rules*, *readability*, *performance* and *security*, given that Stack Overflow code snippets often target a specific user's question or concern [69]. We then developed four research questions (RQ1–RQ4) in Section 2, and presented associated results in the previous section (Section 4). Here we revisit these results and assess their implications in the following four subsections (Sections 5.1 to 5.4), reflecting on our survey of Stack Overflow code snippets' violations against the assessed quality dimensions, as well as the types of violation that are evident in code snippets that are provided by contributors. We then summarise the outcomes of additional correlation analyses performed in Section 5.5.

5.1 Reliability and Conformance to Programming Rules (RQ1)

What is the reliability and conformance to programming rules of code snippets provided in answers on Stack Overflow? We observed that code snippets on Stack Overflow exhibited similar properties, regardless of the existence of violations or not. We also observed that longer code snippets were more likely to produce errors and a larger number of errors. In assessing snippet reliability and conformance to programming rules, we evaluated if code snippets were broken, confusing or prone to runtime errors. We also checked for code snippets' conformance to generally accepted programming rules. Checks considered aspects such as bad comparisons, the use of empty finalizer and return types, coupling, appropriate use of abstract classes, exceptions handling, the correct use of public and private constructors, and so on (refer to D.1 in Appendix D for further details). We observed nearly five violations for each snippet, suggesting that contributors did not focus on the reliability and conformance to programming rules of the code or its ability to compile without errors. That said, if developers in need of help copy

such snippets into their solutions without an awareness of the need to perform refinements, this could lead to the development of poor quality software [68]. This could be particularly troubling, especially if developers are novices, and not cognisant of the reliability and conformance to programming rules aspects considered above. In fact, for many developers, online sources such as Stack Overflow are of utility when they are faced with issues that require knowledge they do not possess [52], and previous evidence has shown that for open source projects which reused Stack Overflow code, only 44% of the snippets were modified prior to their reuse [68]. This brings into focus the potential for the remaining 56% of snippets to have reliability and conformance to programming rules violations. In addition, questions arise around developers' likely understanding of code snippets and their degree of reliability more generally, which in turn brings into question the quality of the software they are likely to develop [37].

Pleasing to observe, however, is the fact that 13,793 code snippets did not contain any reliability or conformance to programming rules violation (out of 50,713 snippets). Such code snippets of course are likely to benefit from extensive review from the many Stack Overflow contributors. The fact that contributors on the platform are rewarded points by other members of the community for their efforts, and especially where answers provided are exceptional, is perhaps of utility for encouraging these complete solutions. For instance, good answers receive votes from community members on Stack Overflow which leads to such answers amassing a score indicative of the number of people in the Stack Overflow community who think the answer is helpful. Notwithstanding some concerns around contributors' gamification of votes [33], users of the Stack Overflow community generally gauge this measure to evaluate answers. We would thus expect those answers without reliability and conformance to programming rules violations to be ranked favourably. Overall, while numerous reliability and conformance to programming rules violations were evident in Stack Overflow code snippets, it is positive that only a very small amount of reliability and conformance to programming rules violations were of the highest priority and the majority were priority three violations. Such a priority is described as "behaviour is confusing, perhaps buggy, and/or against standards/best practices"[21]. This suggests that the code snippets on Stack Overflow may be assessed as generally reliable, which somewhat aligns with the findings of Acar, et al. [2], who found that, as a resource, Stack Overflow produced more functional code than other coding resources. In fact, a majority of violations were categorised in the *Code Style* ruleset which may align with previous outcomes of these authors, as code style of software can often be subjective. For example, software developers may prioritise security or functionality of their code over style or efficiency. Given this reality and the priority of the violations found in Stack Overflow snippets, we believe that developers, while being cautious, could rely on the code snippets found on Stack Overflow to solve their problems. As Parnin and Treude [51] found that 84% of Google search results for a given API contained Stack Overflow results on the first page, it may be fitting that developers turn to Stack Overflow for solutions to their programming problems instead of less scrutinised Q&A portals and blogs.

However, looking at our outcomes in terms of relevance and using a more contextual lens, we see instances of violations ranging from the throwing of a raw exception, the inappropriate use of a method argument or local variable to the instantiating of objects in loops. While not alarming, these issues are noteworthy when considering that novice programmers may reuse Stack Overflow code, thereby inheriting these errors subconsciously. More seasoned developers may also use such code in a hurry, accepting the risks for short term gain, thereby accumulating technical debt. For instance, the throwing of raw exceptions may lead to a lack of clarity, thus extending debugging time. In many cases the Stack Overflow community did not observe these violations, which reinforces the argument that developers should be vigilant when reusing code on Stack Overflow.

5.2 Readability (RQ2)

What is the readability of code snippets provided in answers on Stack Overflow? Our outcomes show that Stack Overflow code snippets contain on average 10.5 readability violations. Readability in this work is defined as conforming to Google's Java conventions and standards (refer to Section 3.1). We anticipated that if Stack Overflow code snippets conformed to such conventions, snippets will be easily understood by the community and, thus, their reuse will lead to good quality maintainable software in the future [27]. At a more granular level, readability checks covered the location of annotations, the prevalence of empty blocks and line separators, good Javadoc conventions, good use of variables, methods and package names, and so on (refer to C.2 in Appendix D for further details). Here

[21] http://pmd.sourceforge.net/pmd-5.0.5/rule-guidelines.html

we see a tendency by the Stack Overflow community to neglect such conventions, again with the potential to affect those reusing snippets in their solutions.

In fact, our evidence points to a much lower level of readability standard for Stack Overflow snippets when compared to their reliability and conformance to programming rules (noted above), suggesting that there was indeed less vigilance around code readability. While this could be somewhat attributed to variance in the use and conformance of coding styles and conventions, readability aspects considered above can be assessed universally (i.e., good use of variables, methods and package names are central to good programming practice). This assessment is supported by the high prevalence of *Indentation, Whitespace, Blocks, Javadoc* and *Naming* readability violations resulting from Stack Overflow snippets. Here we observed that even when Javadoc constructs were used, which is perhaps not expected in code snippets, these at times did not conform to convention. This evidence could be somewhat concerning [52], especially for novice developers seeking help with little awareness of the inadequacy of Stack Overflow snippets.

On the other hand, the high degree of readability violations may be systemic due to the nature of the code snippets that are provided in answers, which are held to be relatively small [69]. Given the likelihood of shorter length code fragments and the neglect of conventions, those reusing such code may be less watchful for this form of inadequacy, particularly the more seasoned members. Of course, the same could be seen with novice members, who are likely to lack awareness. Aspects of readability conventions considered support this assessment, as noted above (e.g., for *Javadoc*). While it may be an expectation for large applications to conform to Javadoc conventions to ensure that programmers understand code, and for software to be maintainable, it is also entirely reasonable that Stack Overflow code snippets are small enough for basic comments to suffice. In fact, these code snippets are often embedded in text that provides explanation outside of the code snippet itself. Squire and Funkhouser [60] found that the ratio of text to code should be approximately 3:1 for answers if developers are to benefit sufficiently from contextual insights. These authors suggest that quality answers (and questions) require sufficient text to explain the code snippet, in order to allow other users to understand the reasoning for the proposed solution. What is sufficient may be debatable, however, and may be linked to the competence of those consuming the knowledge at the time. Also, it does not help that many code snippets and associated explanations are assessed as inadequate by the programming community [65].

Looking at our outcomes in terms of violations' relevance and using more contextual lens, we observed mixed evidence for Stack Overflow community's awareness that readability violations exist. For example, in considering the *EmptyBlockCheck* violation, it was clear that the purpose of the code snippet was to show how to distinguish between object types, and thus the code has empty 'if' blocks. We also observed that the *FallThroughCheck* error was dealt with through recommendations by other Stack Overflow contributors, suggesting that a 'break' statement is needed at the end of each 'case' block (refer to Figure 11.a). Notwithstanding these examples, however, the majority of readability violations went unnoticed by the community (e.g., *AnnotationLocationCheck*, *EmptyCatchBlockCheck* and *AvoidStarImportCheck* violations). These violations may pose challenges to code readability. For instance, the use of an empty 'catch' block (*EmptyCatchBlockCheck*) may lead to confusion about the purpose or importance of the block. Also, there may be usability and security issues associated with not dealing with an identified error, with potential long term consequences [3].

Our findings suggest that some Stack Overflow code snippets may not conform to recommended readability coding conventions, and so developers should be aware of this inadequacy. Those using Stack Overflow code snippets should be willing to properly explore the associated text that is provided as part of answers (and comments) to exhaust all details around the intended solution and associated code. That way, developers could make their own enhancements where context is lacking, and will be able to make informed judgement around the reuse of Stack Overflow code snippets.

5.3 Performance (RQ3)

What is the performance of code snippets provided in answers on Stack Overflow? Our outcomes show that, on average, 0.5 performance violations were found in each code snippet, which provides validation of the Stack Overflow platform. Performance in this work was assessed in terms of the efficiency of the proposed solution. Aspects related to how methods invoke constructors, the allocation of objects, string handling, calling of methods, and so on, are considered under this dimension (refer to D.3 in Appendix D for further details). In fact, the positive

outcome for limited performance issues in Stack Overflow code snippets is supported by our results showing that for those code snippets that contained performance violations, the majority of performance violations were related to the *unread field* (63.5%) category, followed by *unused field* (15.4%). These issues, while perhaps impacting on memory if they are to remain in software code, may not be assessed as critical, particularly given that code snippets are often small in length. That said, if Stack Overflow code snippets are used cumulatively in a code base, such issues could potentially add up, affecting the overall performance of the software and product quality [38].

In fact, evidence indeed confirms that this issue could be challenging for mobile developers, establishing that as much as 5.7% of code published in apps is reused from Stack Overflow [1]. In some contexts, developers have been shown to publish apps largely comprising of reused code [46]. With reuse also extending to teams operating in top-tier software development companies such as Google [10], software released with performance issues could be ubiquitous and troubling for the software engineering community. An example of how catastrophic code reuse can be is illustrated by Bi [11]. This author shows that a piece of Stack Overflow code was used in the NissanConnect EV mobile app, which accidentally displayed a piece of text reading "App explanation: the spirit of stack overflow is coders helping coders". The Stack Overflow answer which has been reused indicates the need to modify the sample string provided[22].

Examining the performance violations for relevance to code snippets and through deeper qualitative lens, we observed that there were noteworthy performance violations in Stack Overflow snippets at times, albeit the community seems aware of some of the violations that were observed. For instance, of the violations that were missed by the community, boxing and then immediately unboxing a primitive value (*Bx_Boxing_Immediately_Unboxed_To_Perform_Coercion*), in order to convert the type of the value, unnecessarily increases compilation time. Under conditions where it is necessary to optimise compilation time, such violations would be an impediment. In the same way, generating a floating-point value and then converting it to an integer involves an unnecessary conversion (*Dm_Nextint_Via_Nextdouble*), which may slow down code and affect its readability. The use of the *number* constructor always creates a new object and, therefore, does not allow the caching of values (*Dm_Number_Ctor*). This uses unnecessary memory and slows down code.

With evidence confirming that anomalies in reused code tend to propagate [40], becoming detrimental to product quality in the long run [36], developers are encouraged to be vigilant when copying Stack Overflow code snippets for reuse. That said, given that, overall, there were limited performance violations in the snippets analysed, developers may also take comfort knowing that code snippets contributed on Stack Overflow generally conform to good performance conventions. Perhaps the Stack Overflow community itself may also encourage the use of good standards to at least maintain the status quo.

5.4 Security (RQ4)

What is the security of code snippets provided in answers on Stack Overflow? Results in this work reveal that there were very few security violations in Stack Overflow snippets. Security violations in this work were assessed in relation to a number of constraints, including the handling of passwords (e.g., evidence of hardcoded or empty passwords), the handling of absolute and relative paths, the use of constants in SQL strings, the invocation of methods in privilege blocks, how references to mutable objects are returned, the protection of fields, methods and classes, and so on (refer to D.4 in Appendix D for further details). When compared to the other three dimensions above (reliability and conformance to programming rules, readability and performance), here we see that there were very few security threats detected. In fact, most security violations observed (85.3%) were in the *mutable static field* category. Such fields may be changed by malicious code and, thus, good security convention demands that they are made private when used.

These outcomes are somewhat in support of the security consciousness of contributors on Stack Overflow, and hold promise for supporting the community's profile. However, it is worth noting that these results are in contrast to the findings of Acar, et al. [2], who conducted manual analysis on a sample of Stack Overflow posts targeted towards Android developers. These authors found that contributors to Stack Overflow produced significantly less secure code when compared to official documentation or books, with just 17% of the Stack Overflow answers studied assessed as containing secured code snippets. It is important to stress that these authors have conducted an experiment involving

---

[22] https://stackoverflow.com/questions/31845450/why-requestwheninuseauthorization-doesnt-prompt-the-user-for-access-to-the-loca

56 respondents' evaluation of four scenarios, which may have included non-compilable code. In addition, the four scenarios were analysed by respondents with varying levels of experience, with 40 of the 56 respondents actually reporting no professional Android development experience. Thus, this issue should be considered when examining the outcomes in our work compared to Acar, et al.'s findings. Beyond these issues however, given that Stack Overflow code snippets are often embedded in text answers, it is pertinent to analyse code snippets that are able to compile in order to evaluate their security, thus providing a fairer assessment. That way, the community would be able to have a more holistic view of security issues, particularly if contributors' intention is not to produce secure code as such, but to provide the basis for which experienced software developers may customise their solutions to make these secure for use.

That said, Fischer, et al. [24] also found that 30.9% of code snippets sampled from Stack Overflow, relating to the Android security API, exhibited security concerns. However, these authors looked at security related code snippets only, suggesting that certain parameters and algorithms posed risks, and therefore were insecure. This is in contrast to our work which considers all Java code snippets irrespective of the intention of the contributor. In fact, while Fischer, et al.'s and Acar, et al.'s studies have found issues with the security of code snippets contributed on Stack Overflow, neither have assessed security in the general context, requiring a large scale coverage of security measures. Our own deeper analysis that focussed on snippets' relevant violations shows that there are security violations in Stack Overflow code snippets which the community is not aware of, albeit these are not numerous. For instance, a hardcoded database password (*Dmi_Constant_Db_Password*) allows anyone with access to the source code to learn the password. This is likely to cause a security breach. If a security manager is installed, creating a classloader requires permissions (*Dp_Create_Classloader_Inside_Do_Privileged*) and, therefore, may require a 'doPrivileged' block (temporarily providing greater privileges in order to perform the task) in order to avoid errors. If a security manager is not installed, creating a classloader may cause a security breach. A similar threat persists for the *Dp_Do_Inside_Do_Privileged* violation. If a security manager is installed, calling a method that requires permissions may need a 'doPrivileged' block (temporarily providing greater privileges in order to perform the task) in order to avoid errors. Storing a reference to an externally mutable object in a (static) field may cause a security breach; the object may be involved in unchecked changes and accessed by untrusted code (*Ei_Expose_Rep2* and *Ei_Expose_Static_Rep2*). These issues were all evident in Stack Overflow code, and there was no evidence that the community understood that these threats existed, suggesting that the wider programming community should exercise caution when snippets are reused.

5.5 Summary

Longer Stack Overflow code snippets possess more reliability and conformance to programming rules and readability violations. We anticipated that contributors would be more prudent in the provision of such code. However, the opposite is observed in our findings, suggesting that Stack Overflow contributors spend little time focussing on limiting code reliability and conformance to programming rules and readability issues. Perhaps many contributors anticipate that code snippets are provided as part of textual answers, and so consumers of such snippets are expected to also peruse the associated text for context and a more complete understanding of the code. This assumption could be counterproductive for the community if members are less knowledgeable, supporting the need for vigilance when using such snippets.

We anticipated that code snippets with violations would be detected by the Stack Overflow community and, thus, such snippets would attract poor scores (downvotes) and limited views, which was somewhat evident in our results (e.g., the answer to Question ID 23721115 was criticised in comments and given a -1 rating due to reliability issues that were detected by the Stack Overflow community). This would discourage those intending to reuse poorly designed code, in favour of Stack Overflow code snippets that attract higher scores. Beyond the example above, we observed that Java-related questions on Stack Overflow that scored higher received more views, and questions that scored higher attracted answers with higher scores. In fact, we also observed a statistically significant negative correlation for *Answer Score* and *Number of Violations* for the reliability and conformance to programming rules, readability and security quality dimensions (rho=-0.09, -0.13 and -0.01 respectively), where those answers that were scored higher had less violations, albeit these are quantitatively small values [18]. This pattern was not evident during our closer examination of *accepted* and *unaccepted* answers; however, although accepted answers tended to score higher, they were not superior in quality to unaccepted answers. Evidence here enforces the position that the

Stack Overflow community pay less attention to contributions with poor quality, and in fact our outcomes here are somewhat against previous evidence of Stack Overflow gamification [33].

Reinforcing this sentiment, and our view above that contributors are likely to be more thoughtful in the provision of longer code snippets on Stack Overflow, code snippets that were longer had less security violations. We also found answers with more code snippets had less reliability and conformance to programming rules and readability violations (rho=-0.25 and -0.27 respectively). These answers also attracted a higher score, which is likely to be due to a more informative answer including more code snippets. These indicators may be useful for the software development community to consider as hints during the assessment of Stack Overflow code snippets.

6. THREATS

The tools used to assess Stack Overflow code snippets have certain limitations. However, these tools are accepted by the software engineering community [34], and have also been validated by academic research [7]. For instance, Ayewah and Pugh [7] evaluated FindBugs using a dataset from Google, finding that this tool was very effective at detecting bugs, and its use was considered to be beneficial for saving time and money for developers. That said, we performed multiple rounds of manual checks to validate our outcomes. We first informally checked the generated output for 100 random errors that were returned by the PMD, Checkstyle and FindBugs tools against the actual code snippets from where the violations were derived. These checks showed that the 100 errors were all traceable in the actual code snippets. We next checked all types of violations returned by the tools for their relevance to code snippets (reliability and conformance to programming rules=191, readability=50, performance=25 and security=14), with inter-rater agreement as measured using Holsti's coefficient of reliability measurement (C.R.) [29], revealing 96.7% agreement initially (first round) and then 100% subsequently (second round). Finally, we conducted a final round of deeper qualitative analysis at the snippet level involving 60 code snippets for each of the four code quality violation categories (i.e., reliability and conformance to programming rules, readability, performance and security). Outcomes from our reliability checks for this analysis reveal 93% agreement initially, with differences resolved on consensus resulting in 100% agreement (refer to Section 3.2.5).

Java is used as a representative sample of code snippets on Stack Overflow. However, different programming languages have different expected conventions, and so require different processing [69]. Therefore, such code snippets would require different criteria for evaluation. It was thus not within the scope of this research project to assess code snippets in answers that were not Java-related, which could be deemed a limitation. However, given Java's popularity and its preference in previous research [22, 62], we believe that by studying this language we provide needed insights for the software engineering community. That said, we have analysed 151,954 Stack Overflow Java code snippets in this work (refer to Section 3.2). Of this total number of code snippets, 101,237 were unparsable by PMD (Reliability and conformance to programming rules violations), 241 code snippets could not be analysed by Checkstyle (Readability violations) and 42,703 code snippets were uncompilable by FindBugs (Performance and Security violations). Our outcomes at most accounted for 50,717 Stack Overflow code snippets. These snippets do not represent the entire population of Java code snippets on Stack Overflow, and thus our outcomes may not be generalizable to all Java posts on Stack Overflow or other programming languages. However, evidence suggests that similar trends may be evident for other languages [9, 48]. We have also established that Stack Overflow data are generalizable across languages and time [43], and we observe a similar pattern of outcomes for aspects of parsable and unparsable Stack Overflow code snippets in this work.

Finally, construct validity aims to verify that a test used to measure a phenomenon actually does so [67]. To ensure construct validity in this work, the individual code snippets' quality dimensions were chosen only if they were considered relevant for evaluating such code [12, 35, 44, 69]. Towards this end, we evaluated a range of code quality dimensions before identifying four code quality attributes (reliability and conformance to programming rules, readability, performance and security) that were deemed suitable for measuring code snippet quality (refer to Section 3.1). In maintaining transparency, rigour and completeness, we also reported on all violations that were detected, before closely examining those violations that were deemed relevant to code snippets, and then providing further outcomes for deeper qualitative analysis. Our reliability checks and formal measures for inter-rater agreement as assessed using Holsti's coefficient of reliability measurement (C.R.) [29] also confirmed excellent agreements (noted above).We thus believe that we have adequately addressed construct validity in this work, and accordingly, have limited this threat.

7. CONCLUSION AND FUTURE RESEARCH

In this study we set out to answer the overarching question: *what is the quality of code snippets provided in answers on Stack Overflow?* We observed that while studies have expressed reservation around the quality of the content provided in Stack Overflow posts, there has been limited effort aimed at evaluating the quality of code snippets on this platform. This is undesirable, as evidence has shown that this platform is used heavily by developers for solving problems during software development. We thus created an agenda to address this gap, and employed an exploratory approach towards understanding the quality of Stack Overflow code snippets. Key to answering our overarching question was defining code snippet quality. Through our evaluation of the code quality body of work, code snippet quality was defined along four dimensions: reliability and conformance to programming rules, readability, performance and security. Appropriate tools were evaluated and selected for our analyses, and Stack Overflow code snippets were then assessed in relation to these dimensions. Our analyses were aimed at evaluating the number of quality violations per snippet, as well as the types of these violations for each of the four code snippet quality dimensions.

Among our findings, we observed that Stack Overflow code snippets contain 4.8 reliability and conformance to programming rules violations on average, and 10.5 readability violations. However, on average there were only 0.5 performance violations in code snippets, with much less security violations observed. The majority of reliability and conformance to programming rules violations were in the "change recommended" category, while *whitespace* readability violations were most prevalent. Performance violations were largely of the *unread field* and *unused field* categories, with the *mutable static field* category comprising the bulk of security violations. We observed that longer Stack Overflow code snippets possess more reliability and conformance to programming rules and readability violations, although such snippets were given less attention by the community, and so may not be reused extensively. Our evidence revealed that Stack Overflow code snippets that were longer had less performance violations, signalling that the contributors providing answers paid more attention to such offerings. Overall, while vigilance is encouraged, Stack Overflow snippets evaluated in this study were not of alarmingly poor quality. In fact, these snippets ranked much higher on performance and security than reliability and conformance to programming rules and readability.

We encourage the replication of our analysis for other Q&A portals that support software development (e.g., Yahoo!Answers). Beyond such avenues for future work, follow up research may also examine Stack Overflow snippets in relation to licensing adherence, which may provide additional insights into the quality of code on Stack Overflow. In line with this proposition, there has been some recent research in this area. For instance, Abdalkareem, et al. [1] have assessed Stack Overflow code reuse in 22 Android applications, while Baltes, et al. [9] and Yang, et al. [70] have investigated the reuse of Stack Overflow code within GitHub projects. We have also studied Stack Overflow code snippet reuse in popular open source projects and within the platform itself [43]. These studies have attempted to understand if software developers provide adequate attributions for copied code. However, these studies have not examined the direction of potential reuse, in terms of whether contributors copy code from GitHub to Stack Overflow without attributions, or vice versa. Such insights would be useful for understanding the degree of license violations prevalent in Stack Overflow code snippets. Another potentially useful project is to further (inductively) explore the differences in quality across accepted and unaccepted answers, which we are planning to conduct. Furthermore, research is required to study the repair effort that is involved with fixing defective Stack Overflow code snippets. These gaps provide avenues for meaningful investigations to further understand Stack Overflow code quality.

8. REFERENCES

1. Abdalkareem, R., Shihab, E. and Rilling, J. On code reuse from StackOverflow: An exploratory study on Android apps. *Information and Software Technology*, *88*. 148-158.
2. Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M.L. and Stransky, C., You Get Where You're Looking For: The Impact Of Information Sources on Code Security. in *Security and Privacy (SP), 2016 IEEE Symposium on*, (2016), IEEE, 289-305.
3. Ahmad, M. and Ó Cinnéide, M., Impact of stack overflow code snippets on software cohesion: a preliminary study. in, (2019), IEEE Press, 250–254.

4. Amir, B. and Ralph, P., There is no random sampling in software engineering research. in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, (2018), 344-345.
5. Anand, D. and Ravichandran, S. Investigations into the Goodness of Posts in Q&A Forums—Popularity Versus Quality. in *Information Systems Design and Intelligent Applications*, Springer, 2015, 639-647.
6. Asaduzzaman, M., Mashiyat, A.S., Roy, C.K. and Schneider, K.A., Answering questions about unanswered questions of stack overflow. in *Proceedings of the 10th Working Conference on Mining Software Repositories*, (2013), IEEE Press, 97-100.
7. Ayewah, N. and Pugh, W., The google findbugs fixit. in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, (2010), ACM, 241-252.
8. Bakota, T., Hegedűs, P., Körtvélyesi, P., Ferenc, R. and Gyimóthy, T., A probabilistic software quality model. in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, (2011), IEEE, 243-252.
9. Baltes, S., Kiefer, R. and Diehl, S., Attribution required: stack overflow code snippets in GitHub projects. in *Proceedings of the 39th International Conference on Software Engineering Companion*, (2017), IEEE Press, 161-163.
10. Bauer, V., Eckhardt, J., Hauptmann, B. and Klimek, M., An exploratory study on reuse at google. in *Proceedings of the 1st international workshop on software engineering research and industrial practices*, (2014), ACM, 14-23.
11. Bi, F. Nissan app developer busted for copying code from Stack Overflow, The Verge, https://www.theverge.com/tldr/2016/5/4/11593084/dont-get-busted-copying-code-from-stack-overflow, 2016.
12. Buse, R.P. and Weimer, W.R. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, *36* (4). 546-558.
13. Buse, R.P. and Weimer, W.R., A metric for software readability. in *Proceedings of the 2008 international symposium on Software testing and analysis*, (2008), 121-130.
14. Campos, U., Smethurst, G., Moraes, J.P., Bonifácio, R. and Pinto, G., Mining rule violations in JavaScript code snippets. in, (2019), IEEE Press, 195–199.
15. CAST. Software Intelligence for Digital Leaders | CAST, https://www.castsoftware.com/, 2019.
16. Cavusoglu, H., Li, Z. and Huang, K.-W., Can Gamification Motivate Voluntary Contributions?: The Case of StackOverflow Q&A Community. in *Proceedings of the 18th ACM Conference Companion on Computer Supported Cooperative Work & Social Computing*, (2015), ACM, 171-174.
17. Chen, F. and Kim, S., Crowd debugging. in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, (2015), ACM, 320-332.
18. Cohen, J. *Statistical power analysis for the behavioral sciences*. Routledge, 2013.
19. Cunningham, W. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, *4* (2). 29-30.
20. di Biase, M., Rastogi, A., Bruntink, M. and van Deursen, A., The delta maintainability model: measuring maintainability of fine-grained code changes. in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, (2019), IEEE, 113-122.
21. Duijn, M., Kučera, A. and Bacchelli, A., Quality questions need quality code: Classifying code fragments on stack overflow. in *Proceedings of the 12th Working Conference on Mining Software Repositories*, (2015), IEEE Press, 410-413.
22. Ercan, S., Stokkink, Q. and Bacchelli, A., Predicting answering times on stack overflow. in *Proceedings of the 12th Working Conference on Mining Software Repositories*, (2015), IEEE Press, 442-445.
23. Ernst, N.A., Bellomo, S., Ozkaya, I., Nord, R.L. and Gorton, I., Measure it? manage it? ignore it? software practitioners and technical debt. in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, (2015), ACM, 50-60.
24. Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M. and Fahl, S., Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. in *Security and Privacy (SP), 2017 IEEE Symposium on*, (2017), IEEE, 121-136.
25. Ginsca, A.L. and Popescu, A., User profiling for answer quality assessment in Q&A communities. in *Proceedings of the 2013 workshop on Data-driven user behavioral modelling and mining from social media*, (2013), ACM, 25-28.
26. Gupta, R. and Reddy, P.K., Learning from Gurus: Analysis and Modeling of Reopened Questions on Stack Overflow. in *Proceedings of the 3rd IKDD Conference on Data Science, 2016*, (2016), ACM, 13.
27. Haefliger, S., Von Krogh, G. and Spaeth, S. Code reuse in open source software. *Management science*, *54* (1). 180-193.
28. Heitlager, I. A Practical Model for Measuring Maintainability–a preliminary report–.
29. Holsti, O.R. *Content Analysis for the Social Sciences and Humanities*. Addison Wesley, Reading, MA, 1969.
30. Holvitie, J., Licorish, S.A., Martini, A. and Leppänen, V., Co-Existence of the'Technical Debt'and'Software Legacy'Concepts. in *QuASoQ/TDA@ APSEC*, (2016), 80-83.
31. Holvitie, J., Licorish, S.A., Spínola, R.O., Hyrynsalmi, S., MacDonell, S.G., Mendes, T.S., Buchan, J. and Leppänen, V. Technical debt and agile software development practices and processes: An industry practitioner survey. *Information and Software Technology*, *96*. 141-160.
32. Hosseini, M., Shahri, A., Phalp, K., Taylor, J. and Ali, R. Crowdsourcing: A taxonomy and systematic mapping study. *Computer Science Review*, *17*. 43-69.
33. Jin, Y., Yang, X., Kula, R.G., Choi, E., Inoue, K. and Iida, H., Quick trigger on stack overflow: a study of gamification-influenced member tendencies. in *Proceedings of the 12th Working Conference on Mining Software Repositories*, (2015), IEEE Press, 434-437.

34. Johnson, B., Song, Y., Murphy-Hill, E. and Bowdidge, R., Why don't software developers use static analysis tools to find bugs? in *Software Engineering (ICSE), 2013 35th International Conference on*, (2013), IEEE, 672-681.
35. Jones, C. and Bonsignour, O. *The economics of software quality*. Addison-Wesley Professional, 2011.
36. Kamiya, T., Kusumoto, S. and Inoue, K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, *28* (7). 654-670.
37. Kan, S.H. *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc., 2002.
38. Knight, J.C. and Dunn, M.F. Software quality through domain-; driven certification. *Annals of Software Engineering*, *5* (1). 293.
39. Kottom, C. Code Quality: Metrics That Matter, 2015.
40. Krinke, J., A study of consistent and inconsistent changes to code clones. in *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE)*, (2007), IEEE, 170-178.
41. Krüger, J., Schröter, I., Kenner, A. and Leich, T., Empirical studies in question-answering systems: a discussion. in *Proceedings of the 5th International Workshop on Conducting Empirical Studies in Industry*, (2017), IEEE Press, 23-26.
42. Letouzey, J.-L., The SQALE method for evaluating technical debt. in *2012 Third International Workshop on Managing Technical Debt (MTD)*, (2012), IEEE, 31-36.
43. Lotter, A., Licorish, S.A., Savarimuthu, B.T.R. and Meldrum, S., Code Reuse in Stack Overflow and Popular Open Source Java Projects. in *2018 25th Australasian Software Engineering Conference (ASWEC)*, (2018), IEEE, 141-150.
44. Lu, Y., Mao, X., Li, Z., Zhang, Y., Wang, T. and Yin, G., Does the Role Matter? An Investigation of the Code Quality of Casual Contributors in GitHub. in *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific*, (2016), IEEE, 49-56.
45. Marshall, M.N. Sampling for qualitative research. *Family practice*, *13* (6). 522-526.
46. Mojica, I.J., Adams, B., Nagappan, M., Dienst, S., Berger, T. and Hassan, A.E. A large-scale empirical study on software reuse in mobile apps. *IEEE Software*, *31* (2). 78-86.
47. Mordal-Manet, K., Balmas, F., Denier, S., Ducasse, S., Wertz, H., Laval, J., Bellingard, F. and Vaillergues, P., The squale model—A practice-based industrial quality model. in *2009 IEEE International Conference on Software Maintenance*, (2009), IEEE, 531-534.
48. Nasehi, S.M., Sillito, J., Maurer, F. and Burns, C., What makes a good code example?: A study of programming Q&A in StackOverflow. in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, (2012), IEEE, 25-34.
49. Nikolaidis, N., Digkas, G., Ampatzoglou, A. and Chatzigeorgiou, A., Reusing Code from StackOverflow: The Effect on Technical Debt. in, (2019), IEEE, 87–91.
50. Oman, P. and Hagemeister, J., Metrics for assessing a software system's maintainability. in *Proceedings Conference on Software Maintenance 1992*, (1992), IEEE, 337-344.
51. Parnin, C. and Treude, C., Measuring API documentation on the web. in *Proceedings of the 2nd international workshop on Web 2.0 for software engineering*, (2011), ACM, 25-30.
52. Ponzanelli, L., Bacchelli, A. and Lanza, M., Leveraging crowd knowledge for software comprehension and development. in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR)*, (2013), IEEE, 57-66.
53. Ragkhitwetsagul, C., Krinke, J., Paixao, M., Bianco, G. and Oliveto, R. Toxic code snippets on stack overflow. *IEEE Transactions on Software Engineering*.
54. Rahman, M.M., Roy, C.K. and Keivanloo, I., Recommending insightful comments for source code using crowdsourced knowledge. in *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, (2015), IEEE, 81-90.
55. Rigby, P.C. and Robillard, M.P., Discovering essential code elements in informal documentation. in *Proceedings of the 2013 International Conference on Software Engineering*, (2013), IEEE Press, 832-841.
56. San Pedro, J. and Karatzoglou, A., Question recommendation for collaborative question answering systems with rankslda. in *Proceedings of the 8th ACM Conference on Recommender systems*, (2014), ACM, 193-200.
57. Shah, C., Oh, S. and Oh, J.S. Research agenda for social Q&A. *Library & Information Science Research*, *31* (4). 205-209.
58. Sommerville, I. Software engineering 9th Edition. *ISBN-10*, *137035152*.
59. Spinellis, D. *Code quality: the open source perspective*. Adobe Press, 2006.
60. Squire, M. and Funkhouser, C., " A Bit of Code": How the Stack Overflow Community Creates Quality Postings. in *System Sciences (HICSS), 2014 47th Hawaii International Conference on*, (2014), IEEE, 1425-1434.
61. Srba, I. and Bielikova, M. A Comprehensive Survey and Classification of Approaches for Community Question Answering. *Acm Transactions on the Web*, *10* (3). 18.
62. Subramanian, S. and Holmes, R., Making sense of online code snippets. in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, (2013), IEEE, 85-88.
63. Suri, H. Purposeful sampling in qualitative research synthesis. *Qualitative research journal*, *11* (2). 63.
64. Treude, C., Barzilay, O. and Storey, M.-A., How do programmers ask and answer questions on the web?: Nier track. in *2011 33rd International Conference on Software Engineering (ICSE)*, (2011), IEEE, 804-807.
65. Treude, C. and Robillard, M.P., Understanding stack overflow code fragments. in, (2017), IEEE, 509–513.

66. Wagner, S., Goeb, A., Heinemann, L., Kläs, M., Lampasona, C., Lochmann, K., Mayr, A., Plösch, R., Seidl, A. and Streit, J. Operationalised product quality models and assessment: The Quamoco approach. *Information and Software Technology*, *62*. 101-123.
67. Westen, D. and Rosenthal, R. Quantifying construct validity: two simple measures. *Journal of Personality and Social Psychology*, *84* (3). 608.
68. Wu, Y., Wang, S., Bezemer, C.-P. and Inoue, K. How do developers utilize source code from stack overflow? *Empirical Software Engineering*, *24* (2). 637–673.
69. Yang, D., Hussain, A. and Lopes, C.V., From query to usable code: an analysis of stack overflow code snippets. in *Proceedings of the 13th International Conference on Mining Software Repositories*, (2016), ACM, 391-402.
70. Yang, D., Martins, P., Saini, V. and Lopes, C., Stack overflow in github: any snippets there? in *Proceedings of the 14th International Conference on Mining Software Repositories*, (2017), IEEE Press, 280-290.
71. Zou, J., Xu, L., Guo, W., Yan, M., Yang, D. and Zhang, X., Which non-functional requirements do developers focus on? an empirical study on stack overflow using topic analysis. in *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, (2015), IEEE, 446-449.

ACKNOWLEDGMENTS

We thank Stack Overflow for granting us access to the data that were analysed in this study. Thanks to the reviewers for their detailed and insightful comments on the early version of this work. This work is funded by a University of Otago Commerce Research Grant Award — accessed through the Otago Business School Research Committee.

APPENDIX A: ANSWER POST DATA SUMMARY

answerId: The unique identifier given to answers on Stack Overflow.
questionId: The unique identifier given to questions on Stack Overflow.
answerScore: The score of the answer, reached by adding the total number of upvotes and subtracting the total number of downvotes.
answerCreationDate: The date and time the answer was first given.
answerBody: The entire body of the answer including the code snippet(s).
questionDate: The date and time the question was first given.
questionScore: The score of the question, reached by adding the total number of upvotes and subtracting the total number of downvotes.
ViewCount: The number of times the question has been viewed.
AnswerCount: The total number of answers that a specific question received.
CommentCount: The number of comments a question received.

answerScore
Minimum: -9
Average: 1.89
Maximum: 3,228

AnswerCreationDate
Earliest: 01/01/2014
Latest: 26/11/2016

answerBody
Minimum length (words): 2
Average length (words): 300
Maximum length (words): 29,038
Minimum no. of code snippets: 1
Average no. of code snippets: 3
Maximum no. of code snippets: 21

questionDate
Earliest: 01/01/2014

Latest: 01/11/2015

questionScore
Minimum: -12
Average: 1.51
Maximum: 1,943

ViewCount
Minimum: 16
Average: 1,521
Maximum: 303,952

AnswerCount
Minimum: 2
Average: 2.73
Maximum: 43

CommentCount
Minimum: 0
Average: 2.58
Maximum: 41

Code Snippets
Minimum LOC: 2
Average LOC: 12
Maximum LOC: 859

Minimum length (characters): 0
Average length (characters): 119
Maximum length (characters): 29,038

Minimum no. of space (“ ”) characters: 0
Average no. of space (“ ”) characters: 25
Maximum no. of space (“ ”) characters: 12,064

Minimum SPA: 1
Average SPA: 1.61
Maximum SPA: 19

APPENDIX B: EXAMPLE ERRORS CAUSED BY CLASS DECLARATION WRAPPER AND JAVA FILE NAMES

The errors associated with the code snippets below are highlighted in red.

'ClassWithOnlyPrivateConstructorsShouldBeFinal' Error (PMD)

```
1  public class C189150{
2  private getEquipmentPredicate(CriteriaBuilder cb, Root<Truck> root) {
3      EquipmentType type = findEquipmentType("not related to this answer");
4      Expression<List<EquipmentType>> equipments = root.get(Truck_.equipmentTypes);
5      Predicate p = cb.isMember(type, equipments);
6      return p;
7  }
8
9  }
```

Error: the 'C189150' class should be 'final'.

'CloneMethodMustImplementCloneable' Error (PMD)

```
1  public class C371214{
2  public Personne clone()
3  {
4      Personne o;
5      try {
6         o = (Personne)super.clone();
7         o.adresse = null;//or just change it however you want here.
8         return o;
9      } catch(CloneNotSupportedException cnse) {
10       cnse.printStackTrace(System.err);
11       throw new RuntimeException();
12     }
13 }
14
15 }
```

Error: the 'C371214' class declaration requires "implements Cloneable".

'CloneMethodReturnTypeMustMatchClassName' Error (PMD)

```
1  public class C371214{
2  public Personne clone()
3  {
4      Personne o;
5      try {
6         o = (Personne)super.clone();
7         o.adresse = null;//or just change it however you want here.
8         return o;
9      } catch(CloneNotSupportedException cnse) {
10       cnse.printStackTrace(System.err);
11       throw new RuntimeException();
12     }
13 }
14
15 }
```

Error: the class name 'C371214' and return type 'Personne' do not match.

‘ProperLogger’ Error (PMD)

```
1 public class C58419{
2 private static final Log LOG_OBJ=LogFactory.getLog(YourClassName.class);
3
4 }
```

Error: the class name ‘C58419’ and class name ‘YourClassName’ parameter do not match.

‘UseUtilityClass’ Error (PMD)

```
1 public class C404764{
2 public static void main(String[] args)
3
4 {
5
6     int i=6;
7     int n=i;
8     int r=0;
9     while(i>0){
10     i--;
11
12 if(i>0 && n%i==0){
13     if (i!=0)
14         r=r+i;
15 }
16
17     }
18     System.out.println(r);
19     if (r==n)
20         System.out.println("its True");
21     else
22         System.out.println("its False");
23 }
24
25 }
```

Error: the ‘C404764’ class should be ‘abstract’.

‘CommentsIndentationCheck’ (Checkstyle)

```
1 public class C244732{
2 //get value like this
3
4   String string = MainActivity.getString();
5
6 }
```

Error: the comment has an indentation level of 0 rather than 2.

‘IndentationCheck’ (Checkstyle)

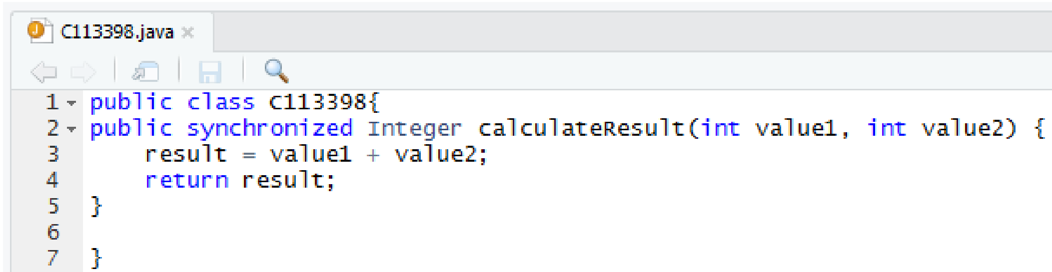
```
1 public class C113398{
2 public synchronized Integer calculateResult(int value1, int value2) {
3     result = value1 + value2;
4     return result;
5 }
6
7 }
```

This code snippet is associated with two errors:

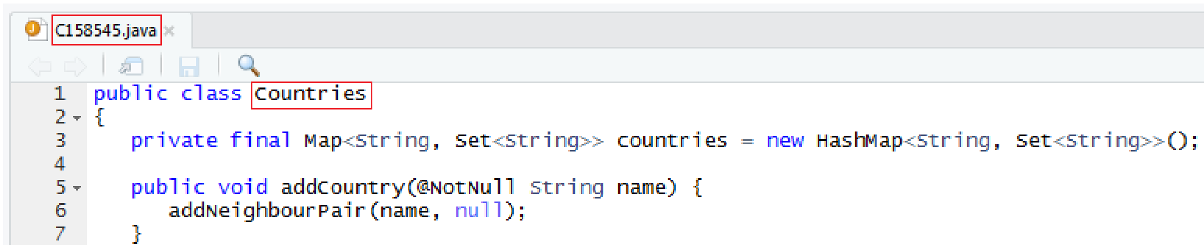
- 'method def modifier' has incorrect indentation level 0, expected level should be 2 (code line 2)
- 'method def rcurly' has incorrect indentation level 0, expected level should be 2 (code line 5)

‘OuterTypeFilenameCheck’ Error (Checkstyle)

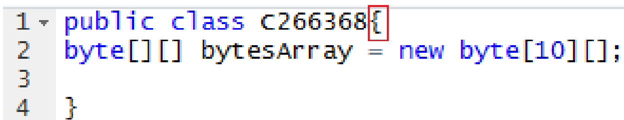
If a code snippet did not contain a class declaration, a class wrapper was added. Therefore, the outer type matches the filename as shown below:



However, if the code snippet already contained an outer type/class declaration, a class wrapper was not added. Therefore, the name of the Java file causes an error. An example code snippet is shown below:



‘WhitespaceAroundCheck’ Error (Checkstyle)



Error: there should be a space between class name ‘C266368’ and ‘{‘.

APPENDIX C: EXAMPLE OUTPUT OF TOOLS

```
<?xml version="1.0" encoding="UTF-8"?>
<pmd version="5.8.1" timestamp="2017-08-26T11:39:23.561">
<file name="C:\Users\Sarah Meldrum\Desktop\pmd-bin-5.8.1\bin\SaveFiles\src\C100196.java">
<violation beginline="1" endline="28" begincolumn="21" endcolumn="1" rule="UseUtilityClass" ruleset="Design" class="C100196"
        externalInfoUrl="https://pmd.github.io/pmd-5.8.1/pmd-java/rules/java/design.html#UseUtilityClass" priority="3">
        All methods are static.  Consider using a utility class instead. Alternatively, you could add a private
        constructor or make the class abstract to silence this warning.
</violation>
<violation beginline="11" endline="11" begincolumn="9" endcolumn="26" rule="SystemPrintln" ruleset="Java Logging" class="C100196"
        method="main" externalInfoUrl="https://pmd.github.io/pmd-5.8.1/pmd-java/rules/java/logging-java.html#SystemPrintln" priority="2">
        System.out.println is used
</violation>
<violation beginline="13" endline="13" begincolumn="9" endcolumn="27" rule="LawOfDemeter" ruleset="Coupling" class="C100196"
        method="main" externalInfoUrl="https://pmd.github.io/pmd-5.8.1/pmd-java/rules/java/coupling.html#LawOfDemeter" priority="3">
        Potential violation of Law of Demeter (object not created locally)
</violation>
<violation beginline="14" endline="14" begincolumn="9" endcolumn="26" rule="SystemPrintln" ruleset="Java Logging" class="C100196"
        method="main" externalInfoUrl="https://pmd.github.io/pmd-5.8.1/pmd-java/rules/java/logging-java.html#SystemPrintln" priority="2">
        System.out.println is used
</violation>
<violation beginline="16" endline="16" begincolumn="9" endcolumn="26" rule="SystemPrintln" ruleset="Java Logging" class="C100196"
        method="main" externalInfoUrl="https://pmd.github.io/pmd-5.8.1/pmd-java/rules/java/logging-java.html#SystemPrintln" priority="2">
        System.out.println is used
</violation>
<violation beginline="20" endline="20" begincolumn="17" endcolumn="30" rule="LawOfDemeter" ruleset="Coupling" class="C100196"
        method="main" externalInfoUrl="https://pmd.github.io/pmd-5.8.1/pmd-java/rules/java/coupling.html#LawOfDemeter" priority="3">
        Potential violation of Law of Demeter (object not created locally)
</violation>
<violation beginline="25" endline="25" begincolumn="5" endcolumn="15" rule="DoNotCallSystemExit" ruleset="J2EE" class="C100196"
        method="main" externalInfoUrl="https://pmd.github.io/pmd-5.8.1/pmd-java/rules/java/j2ee.html#DoNotCallSystemExit" priority="3">
        System.exit() should not be used in J2EE/JEE apps
</violation>
</file>
```

Sample output of PMD

```xml
<?xml version="1.0" encoding="UTF-8"?>
<checkstyle version="8.1">
<file name="C:\Users\Sarah Meldrum\Desktop\SaveFiles\src\C100092.java">
<error line="1" column="21" severity="warning" message="WhitespaceAround: &apos;{&apos; is not preceded with whitespace."
    source="com.puppycrawl.tools.checkstyle.checks.whitespace.WhitespaceAroundCheck"/>
<error line="2" severity="warning" message="&apos;member def type&apos; has incorrect indentation level 4, expected level
    should be 2." source="com.puppycrawl.tools.checkstyle.checks.indentation.IndentationCheck"/>
<error line="2" column="10" severity="warning" message="Member name &apos;d&apos; must match pattern &apos;^[a-z][a-z0-9]
    [a-zA-Z0-9]*$&apos;." source="com.puppycrawl.tools.checkstyle.checks.naming.MemberNameCheck"/>
</file>
<file name="C:\Users\Sarah Meldrum\Desktop\SaveFiles\src\C100093.java">
<error line="1" column="21" severity="warning" message="WhitespaceAround: &apos;{&apos; is not preceded with whitespace."
    source="com.puppycrawl.tools.checkstyle.checks.whitespace.WhitespaceAroundCheck"/>
<error line="2" severity="warning" message="&apos;member def type&apos; has incorrect indentation level 4, expected level
    should be 2." source="com.puppycrawl.tools.checkstyle.checks.indentation.IndentationCheck"/>
<error line="2" column="10" severity="warning" message="Member name &apos;d&apos; must match pattern &apos;^[a-z][a-z0-9]
    [a-zA-Z0-9]*$&apos;." source="com.puppycrawl.tools.checkstyle.checks.naming.MemberNameCheck"/>
</file>
```

Sample output of Checkstyle

```
<ClassStats class="AAAA" sourceFile="N17290.java" interface="false" size="29" bugs="0"/>
<ClassStats class="AAAA$1" sourceFile="N17290.java" interface="false" size="10" bugs="0"/>
<ClassStats class="AAComponent" sourceFile="N290157.java" interface="false" size="11" bugs="0"/>
<ClassStats class="ABall" sourceFile="N26921.java" interface="false" size="20" bugs="1" priority_2="1"/>
<ClassStats class="ACME_bike" sourceFile="N10218.java" interface="false" size="6" bugs="0"/>
<ClassStats class="AClass" sourceFile="N126941.java" interface="false" size="13" bugs="0"/>
<ClassStats class="AColoredItem" sourceFile="N239872.java" interface="false" size="4" bugs="0"/>
<ClassStats class="AComplete" sourceFile="N402583.java" interface="false" size="3" bugs="0"/>
<ClassStats class="AESKeGenFromRSA" sourceFile="N137841.java" interface="false" size="41" bugs="2" priority_3="1" priority_2="1"/>
```


Sample output of FindBugs - File Summary


```
<?xml version="1.0" encoding="UTF-8"?>
<BugInstance type="URF_UNREAD_FIELD" priority="2" rank="18" abbrev="UrF" category="PERFORMANCE">
    <Class classname="ABall">
        <SourceLine classname="ABall" sourcefile="N26921.java" sourcepath="N26921.java"/>
    </Class>
    <Field classname="ABall" name="striped" signature="Z" isStatic="false">
        <SourceLine classname="ABall" sourcefile="N26921.java" sourcepath="N26921.java"/>
    </Field>
    <SourceLine classname="ABall" start="66" end="66" startBytecode="22" endBytecode="22" sourcefile="N26921.java" sourcepath="N26921.java"/>
</BugInstance>
<BugInstance type="SS_SHOULD_BE_STATIC" priority="2" rank="18" abbrev="SS" category="PERFORMANCE">
    <Class classname="AccessBenchmark">
        <SourceLine classname="AccessBenchmark" sourcefile="N43296.java" sourcepath="N43296.java"/>
    </Class>
    <Field classname="AccessBenchmark" name="N" signature="J" isStatic="false">
        <SourceLine classname="AccessBenchmark" sourcefile="N43296.java" sourcepath="N43296.java"/>
    </Field>
    <SourceLine classname="AccessBenchmark" start="2" end="2" startBytecode="8" endBytecode="8" sourcefile="N43296.java" sourcepath="N43296.java"/>
</BugInstance>
```


Sample output of FindBugs - Bug Summary

APPENDIX D: CHECKS/BUGS ASSESSED

D.1 Reliability and Conformance to Programming Rules (PMD[23,24])

All rules (checks) included in the categories below were used to assess reliability and conformance to programming rules.

Reliability (Error Prone)

- AssignmentToNonFinalStatic
- AvoidAssertAsIdentifier
- AvoidBranchingStatementAsLastInLoop
- AvoidCallingFinalize
- AvoidCatchingNPE
- AvoidCatchingThrowable
- AvoidDecimalLiteralsInBigDecimalConstructor
- AvoidDuplicateLiterals
- AvoidEnumAsIdentifier
- AvoidInstanceofChecksInCatchClause
- AvoidLosingExceptionInformation
- AvoidMultipleUnaryOperators
- AvoidUsingOctalValues
- BadComparison
- BeanMembersShouldSerialize
- BrokenNullCheck
- CallSuperFirst
- CallSuperLast
- CheckSkipResult
- ClassCastExceptionWithToArray
- CloneMethodMustBePublic
- CloneMethodMustImplementCloneable
- CloneMethodReturnTypeMustMatchClassName
- CloneThrowsCloneNotSupportedException
- CloseResource
- CompareObjectsWithEquals
- ConstructorCallsOverridableMethod
- DoNotCallSystemExit
- DoNotHardCodeSDCard
- DoNotThrowExceptionInFinally
- DontUseFloatTypeForLoopIndices
- EmptyCatchBlock
- EmptyFinalizer
- EmptyFinallyBlock
- EmptyIfStmt
- EmptyInitializer
- EmptyStatementBlock
- EmptyStatementNotInLoop
- EmptySwitchStatements
- EmptySynchronizedBlock

[23] https://pmd.github.io/pmd-5.8.1/pmd-java/rules/index.html
[24] https://pmd.github.io/latest/pmd_rules_java_errorprone.html

- EmptyTryBlock
- EmptyWhileStmt
- EqualsNull
- FinalizeDoesNotCallSuperFinalize
- FinalizeOnlyCallsSuperFinalize
- FinalizeOverloaded
- FinalizeShouldBeProtected
- IdempotentOperations
- ImportFromSamePackage
- InstantiationToGetClass
- JumbledIncrementer
- JUnitSpelling
- JUnitStaticSuite
- LoggerIsNotStaticFinal
- MisplacedNullCheck
- MissingBreakInSwitch
- MissingSerialVersionUID
- MissingStaticMethodInNonInstantiatableClass
- MoreThanOneLogger
- NonCaseLabelInSwitchStatement
- NonStaticInitializer
- OverrideBothEqualsAndHashcode
- ProperCloneImplementation
- ProperLogger
- ReturnEmptyArrayRatherThanNull
- ReturnFromFinallyBlock
- SimpleDateFormatNeedsLocale
- SingleMethodSingleton
- SingletonClassReturningNewInstance
- StaticEJBFieldShouldBeFinal
- StringBufferInstantiationWithChar
- TestClassWithoutTestCases
- UnconditionalIfStatement
- UnnecessaryBooleanAssertion
- UnnecessaryCaseChange
- UnnecessaryConversionTemporary
- UnusedNullCheckInEquals
- UseCorrectExceptionLogging
- UseEqualsToCompareStrings
- UselessOperationOnImmutable
- UseLocaleWithCaseConversions
- UseProperClassLoader

Conformance to Programming Rules

- Best Practices
    - AbstractClassWithoutAbstractMethod
    - AccessorClassGeneration
    - AccessorMethodGeneration
    - ArrayIsStoredDirectly
    - AvoidPrintStackTrace
    - AvoidReassigningParameters

- AvoidStringBufferField
- AvoidUsingHardCodedIP
- CheckResultSet
- ConstantsInInterface
- DefaultLabelNotLastInSwitchStmt
- GuardLogStatement
- JUnit4SuitesShouldUseSuiteAnnotation
- JUnit4TestShouldUseAfterAnnotation
- JUnit4TestShouldUseBeforeAnnotation
- JUnit4TestShouldUseTestAnnotation
- JUnitAssertionsShouldIncludeMessage
- JUnitTestContainsTooManyAsserts
- JUnitTestsShouldIncludeAssert
- JUnitUseExpected
- LooseCoupling
- MethodReturnsInternalArray
- PositionLiteralsFirstInCaseInsensitiveComparisons
- PositionLiteralsFirstInComparisons
- PreserveStackTrace
- ReplaceEnumerationWithIterator
- ReplaceHashtableWithMap
- ReplaceVectorWithList
- SwitchStmtsShouldHaveDefault
- SystemPrintln
- UnusedFormalParameter
- UnusedImports
- UnusedLocalVariable
- UnusedPrivateField
- UnusedPrivateMethod
- UseAssertEqualsInsteadOfAssertTrue
- UseAssertNullInsteadOfAssertTrue
- UseAssertSameInsteadOfAssertTrue
- UseAssertTrueInsteadOfAssertEquals
- UseCollectionIsEmpty
- UseVarargs

- Code Style
  - AvoidProtectedFieldInFinalClass
  - AvoidProtectedMethodInFinalClassNotExtending
  - ConfusingTernary
  - DontImportJavaLang
  - DuplicateImports
  - EmptyMethodInAbstractClassShouldBeAbstract
  - ExtendsObject
  - FieldDeclarationsShouldBeAtStartOfClass
  - ForLoopShouldBeWhileLoop
  - LocalHomeNamingConvention
  - LocalInterfaceSessionNamingConvention
  - LocalVariableCouldBeFinal
  - MDBAndSessionBeanNamingConvention
  - MethodArgumentCouldBeFinal
  - PrematureDeclaration
  - RemoteInterfaceNamingConvention
  - RemoteSessionInterfaceNamingConvention

    - TooManyStaticImports
    - UnnecessaryFullyQualifiedName
    - UnnecessaryLocalBeforeReturn
    - UnnecessaryModifier
    - UnnecessaryReturn
    - UselessParentheses
    - UselessQualifiedThis
- Design
    - AbstractClassWithoutAnyMethod
    - AvoidCatchingGenericException
    - AvoidDeeplyNestedIfStmts
    - AvoidRethrowingException
    - AvoidThrowingNewInstanceOfSameException
    - AvoidThrowingNullPointerException
    - AvoidThrowingRawExceptionTypes
    - ClassWithOnlyPrivateConstructorsShouldBeFinal
    - CollapsibleIfStatements
    - CouplingBetweenObjects
    - DoNotExtendJavaLangError
    - ExceptionAsFlowControl
    - ExcessiveImports
    - FinalFieldCouldBeStatic
    - GodClass
    - ImmutableField
    - LawOfDemeter
    - LogicInversion
    - LoosePackageCoupling
    - SignatureDeclareThrowsException
    - SimplifiedTernary
    - SimplifyBooleanAssertion
    - SimplifyBooleanExpressions
    - SimplifyBooleanReturns
    - SimplifyConditional
    - SingularField
    - SwitchDensity
    - UselessOverridingMethod
    - UseSingleton
    - UseUtilityClass
- Documentation
    - UncommentedEmptyConstructor
    - UncommentedEmptyMethodBody
- Multithreading
    - AvoidSynchronizedAtMethodLevel
    - AvoidThreadGroup
    - DoNotUseThreads
    - DontCallThreadRun
    - DoubleCheckedLocking
    - NonThreadSafeSingleton
    - UnsynchronizedStaticDateFormatter
    - UseNotifyAllInsteadOfNotify
- Performance
    - AddEmptyString
    - AppendCharacterWithChar

  - AvoidArrayLoops
  - AvoidInstantiatingObjectsInLoops
  - BigIntegerInstantiation
  - BooleanInstantiation
  - ByteInstantiation
  - ConsecutiveAppendsShouldReuse
  - ConsecutiveLiteralAppends
  - InefficientEmptyStringCheck
  - InefficientStringBuffering
  - InsufficientStringBufferDeclaration
  - IntegerInstantiation
  - LongInstantiation
  - OptimizableToArrayCall
  - RedundantFieldInitializer
  - ShortInstantiation
  - SimplifyStartsWith
  - StringInstantiation
  - StringToString
  - TooFewBranchesForASwitchStatement
  - UnnecessaryWrapperObjectCreation
  - UseArrayListInsteadOfVector
  - UseArraysAsList
  - UseIndexOfChar
  - UselessStringValueOf
  - UseStringBufferForStringAppends
  - UseStringBufferLength
- Additional rulesets
  - EmptyStaticInitializer
  - UnnecessaryFinalModifier
- Jakarta Commons Logging
  - GuardDebugLogging
- Java Logging
  - GuardLogStatementJavaUtil
  - InvalidSlf4jMessageFormat

D.2 Readability (Checkstyle[25])

- AbbreviationAsWordInName
- AnnotationLocation
- ArrayTypeStyle
- AtclauseOrder
- AvoidEscapedUnicodeCharacters
- AvoidStarImport
- CatchParameterName
- ClassTypeParameterName
- CommentsIndentation
- CustomImportOrder
- EmptyBlock
- EmptyCatchBlock
- EmptyLineSeparator
- FallThrough

[25] https://github.com/checkstyle/checkstyle/blob/master/src/main/resources/google_checks.xml

- FileTabCharacter
- GenericWhitespace
- IllegalTokenText
- Indentation
- InterfaceTypeParameterName
- InvalidJavadocPosition
- JavadocMethod
- JavadocParagraph
- JavadocTagContinuationIndentation
- LambdaParameterName
- LeftCurly
- LineLength
- LocalVariableName
- MemberName
- MethodName
- MethodParamPad
- MethodTypeParameterName
- MissingJavadocMethod
- MissingSwitchDefault
- ModifierOrder
- MultipleVariableDeclarations
- NeedBraces
- NoFinalizer
- NoLineWrap
- NonEmptyAtclauseDescription
- NoWhitespaceBefore
- OneStatementPerLine
- OneTopLevelClass
- OperatorWrap
- OuterTypeFilename
- OverloadMethodsDeclarationOrder
- PackageName
- ParameterName
- ParenPad
- RightCurly
- SeparatorWrap
- SingleLineJavadoc
- SummaryJavadoc
- TypeName
- UpperEll
- VariableDeclarationUsageDistance
- WhitespaceAround

D.3 Performance (FindBugs[26])

- Bx: Primitive value is boxed and then immediately unboxed
- Bx: Primitive value is boxed then unboxed to perform primitive coercion
- Bx: Primitive value is unboxed and coerced for ternary operator
- Bx: Boxed value is unboxed and then immediately reboxed
- Bx: Boxing a primitive to compare
- Bx: Boxing/unboxing to parse a primitive

---

[26] http://findbugs.sourceforge.net/bugDescriptions.html

- Bx: Method allocates a boxed primitive just to call toString
- Bx: Method invokes inefficient floating-point Number constructor; use static valueOf instead
- Bx: Method invokes inefficient Number constructor; use static valueOf instead
- Dm: The equals and hashCode methods of URL are blocking
- Dm: Maps and sets of URLs can be performance hogs
- Dm: Method invokes inefficient Boolean constructor; use Boolean.valueOf(...) instead
- Dm: Explicit garbage collection; extremely dubious except in benchmarking code
- Dm: Method allocates an object, only to get the class object
- Dm: Use the nextInt method of Random rather than nextDouble to generate a random integer
- Dm: Method invokes inefficient new String(String) constructor
- Dm: Method invokes toString() method on a String
- Dm: Method invokes inefficient new String() constructor
- HSC: Huge string constants is duplicated across multiple class files
- SBSC: Method concatenates strings using + in a loop
- SIC: Should be a static inner class
- SIC: Could be refactored into a named static inner class
- SIC: Could be refactored into a static inner class
- SS: Unread field: should this field be static?
- UM: Method calls static Math class method on a constant value
- UPM: Private method is never called
- UrF: Unread field
- UuF: Unused field
- WMI: Inefficient use of keySet iterator instead of entrySet iterator

D.4 Security (FindBugs)

Security Checks

- Dm: Hardcoded constant database password
- Dm: Empty database password
- HRS: HTTP cookie formed from untrusted input
- HRS: HTTP Response splitting vulnerability
- PT: Absolute path traversal in servlet
- PT: Relative path traversal in servlet
- SQL: Nonconstant string passed to execute or addBatch method on an SQL statement
- SQL: A prepared statement is generated from a nonconstant String
- XSS: JSP reflected cross site scripting vulnerability
- XSS: Servlet reflected cross site scripting vulnerability in error page
- XSS: Servlet reflected cross site scripting vulnerability

Malicious Code Vulnerability Checks

- DP: Classloaders should only be created inside doPrivileged block
- DP: Method invoked that should be only be invoked inside a doPrivileged block
- EI: May expose internal representation by returning reference to mutable object
- EI2: May expose internal representation by incorporating reference to mutable object
- FI: Finalizer should be protected, not public
- MS: May expose internal static state by storing a mutable object into a static field
- MS: Field isn't final and can't be protected from malicious code
- MS: Public static method may expose internal representation by returning array
- MS: Field should be both final and package protected
- MS: Field is a mutable array
- MS: Field is a mutable collection
- MS: Field is a mutable collection which should be package protected
- MS: Field is a mutable Hashtable

- MS: Field should be moved out of an interface and made package protected
- MS: Field should be package protected
- MS: Field isn't final but should be
- MS: Field isn't final but should be refactored to be so